Notional Audit

OPENZEPPELIN SECURITY | DECEMBER 22, 2020

Security Audits

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

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.

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.

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.

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.

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

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.

`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.

`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

`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`.

[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.

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.

`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

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.

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.

`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

- 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.

`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:

- 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`.

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

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.

Introduction

The Notional team asked us to review and audit their smart contracts. We looked at the code and now publish our results.

Note: All links to the project's code in this audit report refer to a repository that is private at the time of writing. They are therefore only accessible to the Notional team.

We audited commit 66ce8f3e529410a5272b7a57c8391fcb80a375fd of the swapnet-protocol/swapnet-lite repository. In scope are all of the contracts inside the directory `/packages/contracts/contracts`, excluding the following contracts:

– `interface/IERC165.sol`
– `interface/MockAggregator.sol`
– `lib/ERC20.sol`
– `lib/ERC1155MockReceiver.sol`

About Notional

Notional is a protocol enabling fixed-term, fixed-rate lending and borrowing on the Ethereum blockchain. The protocol uses a currency called "fCash" to represent an obligation to pay or entitlement to receive tokens at a specified time (or 'maturity'). For example, owning 1000(e18) DAI PAYER fCash tokens represents an obligation to pay 1000 DAI at a specified time, or similarly owning 1000(e18) DAI RECEIVER fCash tokens represents an entitlement to receive 1000 DAI. When the maturity time of a fCash token is reached, users holding fCash can 'settle' their account, thereby converting their fCash balance into a token balance within the protocol.

Borrowing in the protocol must be collateralized by 'deposit currencies', and users that allow their collateral value to drop too low can have their collateral liquidated to cover their borrow.

The protocol allows liquidity providers to add liquidity to each fCash market – where the 2 sides of the market are present-day tokens, and fCash at a specified maturity. This enables lenders and borrowers to then trade cash now for cash later or vice versa, at a rate calculated by the contract.

were separately making functional changes to the protocol to be audited at a later date. This first security-audit of the Notional contracts was the initial step towards reaching the highest levels of code quality and robustness demanded by systems intended to handle large sums of financial assets.

While auditing the contracts we identified numerous vulnerabilities around the handling of ERC20 and ERC777 tokens. Correct handling of tokens and ETH is of vital importance in decentralized financial systems, and is an area that currently needs more attention in the protocol. We also identified vulnerabilities around the mishandling of errors. Although it is a design choice of the Notional team to not revert transactions in certain performance-sensitive situations, it is vital that errors are still handled carefully to prevent errors from propagating into incorrect execution.

Beyond finding vulnerabilities in the code, the Notional team also requested we provide more general feedback on the architecture of the protocol, and on gas usage in functions. Where relevant, some of this advice is incorporated within the issues laid out in the report. However an `Additional Recommendations` section is appended to the report, providing broader feedback on the protocol for the Notional team to consider.

Along with our recommendations for both vulnerabilities and general architecture in this report, further security reviews of the entire protocol are in order, which should help bring the protocol to a production-ready state.

Privileged Roles

The Notional team will initially administer many aspects of the protocol, allowing them to:
– Decide which currencies are accepted,
– Create and remove fCash markets,
– Change governance parameters, and
– Upgrade contracts.

After an initial period, the team intends to transfer this to be a decentralized community-led authority.

Ecosystem Dependencies

collateral in an attempt to pay the account's debts. This audit assumes that these protocols work as intended.

In a move to make the protocol more resilient to the threats posed by such external parties, upon a liquidation Notional compares the prices provided by Uniswap and Chainlink, and reverts if the prices differ by too great a margin. This means that the barrier to attacking the protocol through price manipulation is higher than a protocol that relies on just one price source. However, despite these measures taken to minimize the impact of price manipulation, this is still technically possible and could allow a manipulator to steal money from the protocol. This is a commonly accepted risk with Defi protocols.

Critical

[C01] All USDC can be drained from fCash markets with very little collateral

In the `Escrow` contract, function `_convertToETHWithHaircut` is used to calculate the ETH value of a different currency, and applies a risk haircut to the value. In the final line of the function, a constant value of `Common.DECIMALS = 1e18` is used to calculate the final value in ETH, meaning if `_convertToETHWithHaircut` is executed for any token that does not have 18 decimals, the result is scaled by an incorrect factor of 10.

For example, consider the situation where a user has 0.00000000001 ETH collateral (worth significantly less than 1 cent). The user then calls `FutureCash.takeCollateral` to try to take out USDC in cash in return for 1000 fCash payer tokens. The function calculates the amount of USDC that would get the user, let's say 900 USDC, and updates the market. The user is transferred the 900 USDC, and upserts 1000 fCash payer tokens. Whilst upserting cash payer tokens into the user's portfolio, `freeCollateral` is executed to ensure that the user's account won't be left undercollateralized.

The call leads to `Portfolios.freeCollateralView`, where all currencies are converted into equivalent ETH values. However when USDC balances are converted into their equivalent ETH values, the ETH value calculated is off by a factor of `1e12`.

calculates the equivalent ETH value as `((1000 * 1e6) * (0.002235 * 1e18)) / 1e18`, which is `2235000 WEI` or `0.000000000002235 ETH`.

This means that large USDC cash payer balances can be collateralized with tiny amounts of ETH. In the example above, the user would be able to immediately withdraw their 900 USDC leaving the 0.00000000001 ETH to collateralize their -1000 USDC cash payer balance. The user can repeatedly call `takeCollateral` with `futureCashAmount = G_MAX_TRADE_SIZE` until they have withdrawn all of the USDC collateral from the market. They can then call `Escrow.withdraw` to withdraw all the USDC collected. `Escrow.withdraw` uses the same free collateral calculation and will approve withdrawing the USDC from an account with just a tiny amount of ETH.

Besides `withdraw` and `takeCollateral`, `freeCollateralView` is also used to calculate the free collateral of accounts during liquidation, and settling cash balances. Consider adding a function `getDecimals` that gets the underlying decimals of an ERC20 token, and then replacing `Common.DECIMALS` on L1138 of Escrow with a call to `getDecimals(base)`. More generally, consider updating token calculations throughout the Notional Protocol to use `getDecimals` instead of `Common.DECIMALS` – see issue **[H01] Uniswap-Chainlink rate difference always reverts for certain currencies** for another issue caused by incorrect handling of decimals.

Update: *Fixed in pull request #47. A change was made to line 1136 of `Escrow.sol`, but it has the same effect as the one suggested.*

[C02] ERC777 deposits can be credited twice

The `deposit` function within `Escrow.sol` allows users to transfer an ERC20 token while crediting their account for the amount of the deposit. Similarly, ERC777 tokens can be deposited by calling the `send` function of the ERC777 contract, which then calls the `tokensReceived` hook, which in `Escrow.sol` will increment the user's balance for that token.

However, there is no check within `deposit` that prevents the call to `transferFrom` if `token` is an ERC777 compliant token. Additionally, the OpenZeppelin ERC777 implementation has an implementation of `transferFrom` which calls `tokensReceived` after a transfer. So,

again within `tokensReceived` for a total of `2*X` tokens being credited to their account. After this, a user could `withdraw` `2*X` tokens, receiving "free money" (unless there are not enough tokens in the contract to do so).

Consider implementing some check which disallows ERC777 tokens from being transferred via the `deposit` function. Alternatively, consider only listing ERC777 tokens which do not implement a `transferFrom` which calls `tokensReceived`. In general, all tokens should be vetted thoroughly before listing, adhering to a strict set of rules which exist to prevent damage to the protocol.

Update: *Fixed in pull request #49 – users can no longer deposit ERC777 tokens via the* `deposit` *function.*

[C03] Borrow rates can be severely miscalculated

Within `FutureCash.sol`, the `_getExchangeRate` function will return `0` if it encounters various errors. Within `_tradeCalculation`, the value of `tradeExchangeRate` is set to the returned value from calling `_getExchangeRate`. After this, if `futureCashAmount <= 0`, `tradeExchangeRate` will be set to `tradeExchangeRate - fee`. Since both `fee` and `tradeExchangeRate` are `uint` values, this will cause `tradeExchangeRate`'s value to underflow and become close to the max value for a `uint32`.

On line 944, `collateral` is calculated using `tradeExchangeRate`, which may now hold an erroneous value. Since `tradeExchangeRate` has underflowed, it may be much higher than a correct value would be. Here it is the divisor, so a higher `tradeExchangeRate` implies a lower value for `collateral` than under normal circumstances.

Then, on lines 956 and 957, new `totalFutureCash` and `totalCollateral` values are calculated for the market, and are subsequently returned to the function calling `_tradeCalculation`. Within `_updateMarket`, the returned values of `_tradeCalculation` are used to update the market involved, applying these erroneous values to the system and effecting the market overall.

will be called with the negative value of `futureCashAmount` passed in. This now-negative value will be passed into `_tradeCalculation`. `tradeExchangeRate` will be set to the result of `_getExchangeRate`, with `futureCashAmount` passed in. Within `_getExchangeRate`, `numerator` will equal `0`, and this will cause `proportion` to equal `0`. `proportion` will be passed into `_abdkMath`, which will cause it to return `(0, false)`. Finally, this results in `_getExchangeRate` returning `0`, and setting `tradeExchangeRate` to `0`. If we assume `fee = 200`, then `tradeExchangeRate` will be set to `4294967095` or roughly `4.29e9`. Assuming a `futureCashAmount` of `1000e18` (1000 DAI) and an `INSTRUMENT_PRECISION` of `1e9`, `collateral` will be set to roughly `232.8e18`. What this effectively means is that a user can receive 1000 “future DAI” for ~233 current DAI. This is likely far more than a reasonable interest rate the market would allow.

Consider implementing SafeMath’s `sub` in the application of `fee` to `tradeExchangeRate` to ensure that any underflows are not permitted. Additionally, consider adding checks for `0` returned values when they correspond to errors and reverting when appropriate. This helps to follow the “fail early” design principle. See related issue **[H03] `_getImpliedRate` does not handle errors**.

Update: *Fixed in pull request #54. Checks are implemented to prevent under/overflow rather than using SafeMath.*

High

[H01] Uniswap-Chainlink rate difference always reverts for certain currencies

In the `Escrow` contract, function `_checkUniswapRateDifference` compares the rate from Uniswap and Chainlink, and reverts if the rates are not similar to protect users from bad rates. However `_checkUniswapRateDifference` assumes that the tokens involved, and the Chainlink oracle price have 18 decimals in their value – in cases where one or both of these is not true, the function may revert even when the prices are identical.

– DAI/USD

A few examples of tokens that do not have 18 decimals are:

– WBTC

– USDC

– TUSD

Consider the example where the USDC/ETH rate is being checked using `_checkUniswapRateDifference`, where USDC has 6 decimals and the USDC/ETH oracle has 18. Uniswap swapped `1 ETH for 321 USDC`, and the Chainlink oracle has a price of `1 USDC is 0.00311526479 ETH` (with 18 decimals) – these 2 prices are identical. In the function `uniswapImpliedRate` is calculated as `(1e18 * 1e18) / 321e6`, which is `0.00311526479 * 1e30`, whereas the Chainlink oracle returns `answer` as `0.00311526479 * 1e18`. This difference of value causes the function to revert even with identical prices. A similar situation occurs when both tokens have 18 decimals, but the Chainlink oracle has fewer decimals.

Two other functions that rely on tokens or oracles having 18 decimals are: `_purchaseCollateralForLocalCurrency` and `_exchangeRate`. Consider adding a field to the `ExchangeRate` struct that holds the number of decimals of the `rateOracle`, and adding the `getDecimals` function outlined in **[C01] All USDC can be drained from fCash markets with very little collateral** to get the number of decimals of an ERC20 token. These decimal values should then be used throughout the protocol to correct calculations using Chainlink oracles and ERC20 tokens. Also, see related issue **[H02] Chainlink rates may need to be inverted**.

Update: *Partially fixed in pull request #48. The computation is still incorrect if* `er.rateDecimals` *causes* `rateDiff` *to have a different number of decimals than* `Common.DECIMALS`*. Notional state that this logic is removed, which OpenZeppelin will verify in an upcoming 2nd audit.*

[H02] Chainlink rates may need to be inverted

In some cases, the oracle may not exist for a given pair. For instance, imagine a uniswap trade of WBTC for DAI. This makes sense because WBTC is a collateral, and DAI is a cash or `localCurrency` token. The value `localCurrencyPurchased` is the amount of DAI tokens received from a Uniswap trade, and at a price of roughly 11000 USD/BTC, would be 11000 DAI for 1 WBTC sold. So, `localCurrencyPurchased = 11000e18`. Meanwhile, `collateralSold` should equal `1e8` to represent 1 WBTC token. Both of these values will be passed into a call to `_checkUniswapRateDifference`.

Within `_checkUniswapRateDifference`, `uniswapImpliedRate` will be calculated as `1e8 * Common.DECIMALS / 11000e18`, where `Common.DECIMALS = 1e18`. So, the calculation result is `9090`. Intuitively, this value represents the number of WBTC units ("W-satoshis") which can be purchased for 1 DAI. This value will be compared with the chainlink oracle's `answer`, which will come from the chainlink BTC-USD feed. This feed provides `latestAnswer` in terms of the USD price of BTC, scaled up by a factor of 1e8. So, for a BTC price of `11000 USD`, the `answer` will be `11000e8`. This is very different in its order of magnitude versus the `uniswapImpliedRate`, which is discussed in more detail in issue **[H01] Uniswap-Chainlink rate difference always reverts for certain currencies**. It also has the important difference that instead of measuring WBTC units per DAI, it is measuring DAI units per WBTC (scaled by some orders of magnitude).

Both the difference in the measured units (measuring WBTC in DAI units versus measuring DAI in WBTC units) and the difference in order of magnitude should be taken into account, or the comparison is likely to revert. Even if the comparison does not revert, it is useless in principle to compare two values which do not represent the same thing. Since a chainlink oracle does not exist for USD in terms of BTC, the `answer` from the oracle must be processed to invert the ratio (disregarding the fact that the correct scaling will need to be applied as well). Consider adding a `bool` to the `ExchangeRate` struct which flags whether a chainlink rate will need to be inverted. Considering the recommendation of **[H01]**, make sure to define the oracle's decimals as the number of decimals either before or after the inversion, and clearly state this in the documentation and comments. Before adding a new oracle, consider implementing thorough testing with real-world values for the pairs that will be created.

Update: *Fixed in pull request #48.*

The `_getImpliedRate` function calls `_getExchangeRate` and uses it in calculations. In various failure cases within `_getExchangeRate`, the value `0` will be returned.

If `0` is returned, within `_getImpliedRate` the return value will underflow due to an unprotected subtraction, resulting in a much higher "implied rate" than in a non-error case.

Importantly, this value will be used within `_tradeCalculation` to set a market's `lastImpliedRate` and within `_getNewRateAnchor`. This can cause future `rate` calculations to be too high, resulting in a return of `0` from `getExchangeRate` and perpetuating the problem. Miscalculations of `.lastImpliedRate` can also throw off future `rateAnchor` calculations again perpetuating the problem.

Consider calling `_getImpliedRateRequire` which uses `sub` to protect against underflow if `_getExchangeRate` returns `0`, rather than `_getImpliedRate`. Alternatively, if it is not desired to revert, consider determining a safe "max" rate, and branching within `_getImpliedRate` whenever `_getExchangeRate` returns `0`, to avoid returning an erroneous value by subtracting from `0`. See related issue **[C03] Borrow rates can be severely miscalculated**.

Update: *Fixed in pull request #54.*

[H04] Rounding when performing `div` causes inconsistent balances

In `FutureCash.tradeLiquidityToken`, lines 786-788 calculate a number of `tokensToRemove` based on an amount of `collateralRequired`. This `tokensToRemove` is plugged into `_settleLiquidityToken` to remove the collateral and fCash from the market. The returned `collateral` from `_settleLiquidityToken` is ignored – as it's assumed that it must be identical to the `collateralRequired` above. However due to the fact that Solidity truncates when dividing, `collateralRequired` and `collateral` can end up with different values. The market's `totalCollateral` gets reduced by `collateral`, but the market's `currencyBalance` is eventually reduced by `collateralRequired`. This means that a market's `totalCollateral` can end up with a higher value than the true balance of the market.

– `totalCollateral` = 1000

The value of `tokensToRemove` is calculated as `(720*90)/1000`, which is 64. The value of `collateral` in `_settleLiquidityToken` is calculated as `(1000*64)/90`, which is 711. The market's `totalCollateral` is reduced by 711, leaving a balance of 289, whereas the market's actual currency balance gets reduced by 720 to 280.

This balance discrepancy will get amplified every time such a rounding error occurs. This discrepancy in balances can end up causing a number of problems in the protocol, including:
– A higher sum of currencyBalances than currency actually available for withdrawal
– Incorrect calculation of the amount of collateral given to a liquidity provider when they withdraw
– Incorrect amounts of liquidity and cash payer tokens being minted to new liquidity providers

Consider uncommenting the result returned from `_settleLiquidityToken`, and using this as the return value from `tradeLiquidityToken`, instead of the pre-computed `collateralRequired`.

Update: *Fixed in pull request #51.*

[H05] Wrong value in call can cause money to be incorrectly withdrawn from reserves

In `_settleCashBalance` within `Escrow.sol`, a call to `_convertToETHWithHaircut` is made using the value `settledAmount`. However, `settledAmount` is declared within the function giving it a value of `0` initially. The only place `settledAmount` is modified is within the `if` portion of the branch on line 730. However, the call to `_convertToETHWithHaircut` is within the `else` section of the same structure, and since `settledAmount` is not modified within that branch, it will still be `0` when `_convertToETHWithHaircut` is called. This will cause the calculation of the `payer`'s `freeCollateral` to be incorrect, leading to further liquidation of the `payer`'s assets, and may end up causing the protocol to take currency out of the protocol's reserves when it isn't needed.

It appears that the intention was to use the value `valueToSettle - localCurrencyRequired` in the call to `_convertToETHWithHaircut`. So, consider

Update: *Fixed in pull request #52.*

Medium

[M01] `ERC777` tokens may not implement `ERC20` functions

EIP 777 states that ERC777 tokens "MAY implement both ERC20 and ERC777 in parallel", but not that they must do so. Thus, some ERC777 tokens may not implement ERC20 standard functions such as `transfer` or `transferFrom`.

Within `Escrow.sol`, the `withdraw` function uses a call to `transfer` to send tokens to a withdrawing user. There is no other way for a user to withdraw tokens aside from this function. If an ERC777 token does not implement a `transfer` function, the call to `transfer` within `withdraw` will fail (or possibly behave unexpectedly, if a `fallback` function exists in the ERC777 contract) and the user will not be able to extract their tokens.

Consider adding a process to vet ERC777 tokens to ensure that they implement a `transfer` function which behaves like an ERC20 transfer. Alternatively, consider implementing separate functionality to call the ERC777 `send` function when withdrawing ERC777 tokens.

Update: *Fixed in pull request #49. The `withdraw` function now calls `send` for ERC777 tokens.*

[M02] Casting between types without overflow checks

Throughout the codebase, many instances of both implicit and explicit casting between types exist, with both having the potential to impart overflow errors when downcasting or casting between `int` and `uint`.

Some examples of such behavior can be found on:
– line 382 of `FutureCash.sol` where a `uint256` value is downcast to `uint128`.
– line 466 of `FutureCash.sol` where a `uint256` value is downcast to `uint128`.
– line 1098 of `Escrow.sol` where a `uint256` is cast to `int256`.

Update: *Partially fixed in pull request #54. Line 1002 of `Escrow.sol` still uses casting in a way that allows for overflows, however Notional state this logic is removed. OpenZeppelin will verify this in an upcoming 2nd audit.*

[M03] Math issues in SafeInt256.sol

There are a number of issues in `SafeInt256.sol`, whereby the functions do not return the expected result for a given input. A few examples of such issues are outlined below.

`function mul`

`mul` does not correctly check for overflowing results, meaning that `mul(-1, -2**255)` returns `-2**255`. The expected result of this should be `2**255`, which is too large for an `int256`, and therefore it should revert.

`function abs`

Similarly to `mul`, `abs` does not check for an overflowing value, meaning that `abs(-2**255)` returns `-2**255`. The expected result of this should be `2**255`, which is too large for an `int256`, and therefore it should revert.

`function div`

`div` has a check `y>0` for the second parameter of the function, meaning that for any negative integer, the function would revert with error `DIVIDE_BY_ZERO`. Expected behaviour would be for `div` to be able to divide by a negative number without throwing.

Consider using OpenZeppelin's `SignedSafeMath` library for `mul` and `div` instead of the versions implemented in `SafeInt256.sol`. To correct a potential overflow in `abs`, consider changing line 49 to use the `neg` function, which includes an overflow check.

Update: *Fixed in pull request #50.*

[M04] Unchecked ERC20 return values

instead return `false`.

In `Escrow.sol`, the functions `deposit` and `withdraw` do not check the return value of the calls to `transferFrom` and `transfer` respectively. This means that for a token that returns a `bool` without throwing upon a failing transfer, a user could call `deposit` and have their `currencyBalance` incremented while not actually depositing any tokens.

Due to the fact that some tokens do not return a boolean, and some tokens do not throw upon failure, the code should check for both cases in the event that a token transfer fails. Consider instead using the `safeTransfer` and `safeTransferFrom` functions from the OpenZeppelin library `SafeERC20`. This library safely handles ERC20 tokens that behave in different ways, and ensures that all calls revert on failure, whether or not the underlying token does. See related issue **[L05] Token fees are not accounted for**.

Update: *Fixed in pull request #49. `SafeERC20` is now used for token transfers and approvals.*

[M05] Cash settling during low liquidity deletes cash receiver tokens

During the execution of `settleCashBalance`, the flow can lead to the execution of `_attemptToSettleWithFutureCash`, which itself executes `Portfolios.raiseCollateralViaCashReceiver`. When a cash receiver token is found in the user's portfolio, `FutureCash.tradeCashReceiver` is called to sell the full amount of fCash receiver tokens in exchange for collateral. The receiver tokens are then removed from the user's portfolio by creating the same number of payer tokens to offset the receiver tokens.

However if the FutureCash market has low liquidity, with `market.totalFutureCash` less than the fCash amount being sold, then only `market.totalFutureCash` tokens are sold, not the full amount. This is not communicated back to the calling function, and so the full amount of receiver tokens are removed from the user's account even though not all of them were sold for collateral.

Consider adding a second return parameter to `FutureCash.tradeCashReceiver` to return the number of fCash receivers actually sold. The number of receiver tokens removed from the user's account can then be adjusted to reflect this trade accurately.

[M06] Not all calculations use SafeMath

Calculations that do not check for under/overflow can easily cause severe issues in Solidity. Throughout the majority of the code, SafeMath.sol is used to ensure that any calculations that could face such issues revert instead of under/overflowing. However as seen in **[C03] Borrow rates can be severely miscalculated**, and **[H03] `_getImpliedRate` does not handle errors**, some lines of code do mathematical calculations without using SafeMath which has lead to issues.

Besides the named issues above, other lines of code that do not check for math errors and therefore risk under/overflow include:
– Common.sol line 155
– FutureCash.sol line 990 and line 1054
– Escrow.sol line 907

Consider using using SafeMath.sol for all calculations, except those that are previously checked for errors using an `if` or `require`.

Update: *Fixed in pull request #54.*

[M07] `liquidate` can leave accounts undercollateralized despite assets being available

When calculating the `freeCollateral` for an account, the account's `cashLadder` is constructed for each currency. A cash ladder is essentially a list of net balances for each maturity in a collateral, where each maturity can be thought of as a rung in the ladder. In the construction of a cash ladder, positive assets in a maturity are used to offset any `payer` tokens within the same maturity. These 'positive assets' can include the `receiver` tokens and collateral that liquidity holders have claim over a share of.

When free collateral is then calculated from these cash ladders, any rungs with a net positive balance are ignored, and the negative balances are summed. If these negative balances aren't collateralized by enough collateral or 'current day' cash, then the account is considered to be under-collateralized and can be liquidated.

bring this back to 0.

The first step the protocol takes to raise the currency needed to re-collateralize an account is to trade any liquidity tokens the user holds. When trading the liquidity tokens, any currency gained in return is used to directly offset the negative currency requirement. If the total currency requirement is raised, then execution stops, thinking that the account is collateralized again.

However, due to the fact that the positive value associated with liquidity tokens was already included in the cash ladder calculations for maturities with a net-negative currency requirement, the currency raised trading these liquidity tokens during liquidation will not actually be offsetting the currency requirement. This will cause liquidation to stop, thinking that an account is re-collateralized, when it is not. Users may then have to repeatedly call `liquidate`, liquidating tokens that do not need to be liquidated, until liquidation reaches an asset that does offset the negative currency requirement and re-collateralize the account.

Consider altering the liquidation procedure so that the only assets traded are those that will offset the negative currency requirement, and re-collateralize the account

Update: *Acknowledged. Notional's statement for this issue:*

> the fix for this issue is beyond the scope of the audit and a review is planned for a subsequent audit

[M08] Loops may consume too much gas

There are many instances in the code of large `for` loops, which are known to consume high amounts of gas when in smart contracts. For example, the `_liquidate` function runs through many `for` loops, such as:
– the loop within `_tradePortfolio`, called twice within a `_liquidate` call
– the loop within `_quickSort`, called three times within a `_liquidate` call
– the loop within `_settleAccount`

If smart contract execution reaches the block gas limit due to too much looping, the `liquidate` function may be unable to execute, preventing liquidations on certain accounts. Notably, the likelihood of this increases as the account in question has more assets in their portfolio. So, by increasing their number of assets, an account can prevent their liquidation.

Since the Notional team has indicated a desire to increase gas efficiency in their codebase, and since this section of the code is particularly gas-intensive, consider refactoring the liquidation code to use fewer loops. Consider creating tests which account gas usage both before and after the refactor, to ensure and help quantify the efficiency increase. Finally, consider imposing a limit of portfolio size based on a target max gas usage, to prevent the liquidation process from getting stuck.

Update: *Partially fixed in pull request #56. The changes implemented in this pull request pertain to setting a max number of assets. Tests should also be created to ensure that flows which call* `_upsertAsset`*, such as the liquidation flow, can handle situations where a user's portfolio is full. Notional state this logic is fixed for their upcoming 2nd audit.*

[M09] Owner of `Portfolios` can disable markets

By calling `updateFutureCashGroup` with a `numPeriods` value of `0`, the `owner` of `Portfolio.sol` can cause the call to `FutureCash.setParameters` to set `G_NUM_PERIODS` to `0` for that FutureCash market.

This will then cause every call to `_isValidBlock` to `revert`, preventing successful calls to `addLiquidity`, `removeLiquidity`, `takeCollateral`, and `takeFutureCash`.

Since this change effectively shuts down the market instantly, consider adding a time-delay mechanism which allows users a grace period between the signaled intent to close the market, and the actual closing of the market. Alternatively, if the ability to immediately close the market is desired, consider informing users expicitly of this admin power in the documentation, and providing a means to allow users to recover their funds

Low

[L01] `INSTRUMENT_PRECISION` is not fixed

Through discussions with the Notional team, we understand the value of `INSTRUMENT_PRECISION` is intended to be `1e9`. Also, it is supposed to be unchangeable once set.

The check on line 938 of `FutureCash.sol` uses `INSTRUMENT_PRECISION` to check whether the `tradeExchangeRate` calculated is greater than a rate of `1`. This implies that a rate of `1` is represented by the value of `INSTRUMENT_PRECISION`. Within the calculation of the rate, the usage of hardcoded values shows that regardless of the value of `INSTRUMENT_PRECISION`, `tradeExchangeRate` will be expressed as a value scaled by `1e9`. For instance, part of the calculation in `_adbkMath` scales a number by `PRECISION_64x64` which is the value `1e9`. Later on, the returned value from `_abdkMath` has `LN_1e18` subtracted from it. To achieve the desired mathematics, `LN_1e18` represents the natural log of `1e18`, scaled by `1e9`. Since both `PRECISION_64x64` and `LN_1e18` are hard-coded values, and both are scaled by `1e9`, no matter what the resulting value for `tradeExchangeRate` will be scaled by `1e9`. Thus, setting `INSTRUMENT_PRECISION` to anything besides `1e9` will break the check on line 938.

Consider hard-coding the value of `INSTRUMENT_PRECISION` to `1e9` rather than setting it manually.

Update: *Fixed in pull request #61.*

[L02] `_listCurrency` function has no input checks

`Escrow.sol`'s `_listCurrency` function provides no checks to see if `token` has already been listed. The comments explain that this function relies on governance to call this function correctly.

If this function is called incorrectly, it could lead to a currency having 2 different IDs in `currencyIdToAddress`, and cause `freeCollateral` calculations to double count the

Consider adding a check that `addressToCurrencyId[token] == 0 && token != WETH` within `_listCurrency` to prevent mistakes when calling the function.

***Update**: Fixed in pull request #49.*

[L03] `div` should always be performed as late as possible in calculations

Other issues in this report have highlighted truncation errors resulting from the `div` function, or the `/` operator. In general, all divisions should occur as the final steps of a calculation, to minimize the compounding effects of truncation. Within `Escrow.sol`, on lines 991 and 998, two `.div` operations are performed, with a `.mul` operation between them.

Although performing these divisions intermittently can help reduce risk of overflow errors, with a `rate` and a `discountFactor` value on the order of `1e18`, and a `Common.DECIMALS` value of `1e18`, `localCurrencyPurchased` (on line 991) or `collateralBalance` (on line 998) would need to be on the order of `1e41` to cause an overflow. Given that values will be in wei or token units, a value of `1e41` translates to roughly `1e23` (100 Sextillion) whole ETH or DAI. Overflow is therefore not a risk in these calculations, and calculation accuracy can instead be prioritized.

To favor more accurate mathematics, consider moving the `div` operations to the end of the calculations in the identified spots. Alternatively, if this is desired to prevent a possible overflow (perhaps in the case of a token with a very high number of decimals), leave a comment explaining the possibility of this case.

***Update:** Fixed in pull request #48.*

[L04] `_checkUniswapRateDifference` may not use highest percentage difference

The `_checkUniswapRateDifference` function calculates `rateDiff` as a proportion of `answer`, the Chainlink oracle rate. Meanwhile, the `uniswapImpliedRate` is based off of the input and output values from a Uniswap trade. If the `rateDiff` is too high, execution will revert.

can see, when calculated as a proportion of the `uniswapImpliedRate`, the `rateDiff` may be higher than in the current case.

Since the code is supposed to revert when the rate difference is too high, both rate difference proportions should be checked. Consider calculating the rate difference as a proportion of the lower rate, which will guarantee that the `rateDiff` is always higher than the other way around.

Update: *Fixed in pull request 48.*

[L05] Token fees are not accounted for

In `Escrow.sol`, within the `deposit` function, a user's currency balances will be incremented by the same amount as is passed to the `transferFrom` function to perform the transfer.

Some ERC20 tokens have the capability to charge fees, such that the amount transferred is not the same as the amount passed to the `transfer` or `transferFrom` function. One notable example is the popular USDT token, where the recipient of a transfer receives `value - fee`. Currently no fee is charged, but if it were, this would lead to the total currency balances being different from the actual amount of currency held by the contract, and could therefore lead to insolvency.

Consider vetting tokens for a lack of fee-charging capabilities. Alternatively, consider implementing calculations to see how much token balances have increased before crediting an account. See related issue **[M04] Unchecked ERC20 return values**.

Update: *Fixed in pull request #49.*

[L06] `futureCashGroups` parameters may not match `FutureCash` parameters

Within `FutureCash.sol`, the `setParameters` function allows parameters to be set for the contract on the condition that `FUTURE_CASH_GROUP == 0`. Once `FUTURE_CASH_GROUP` has been set to a non-zero value, any future calls to this function will not effect the parameters on lines 115 to 118.

`FutureCash.sol`. Notably, `.periodSize` within `Portfolios.sol` matches `G_PERIOD_SIZE` within `FutureCash.sol`, and `.precision` within `Portfolios.sol` matches `INSTRUMENT_PRECISION` within `FutureCash.sol`. However, as explained above, the `updateFutureCashGroup` function may update the values in `Portfolios.sol`, but `setParameters` may not make the corresponding changes in `FutureCash.sol`.

To prevent mismatches between the `futureCashGroups` mapping in `Portfolios.sol` and the `INSTRUMENT_PRECISION` and `G_PERIOD_SIZE` variables within `FutureCash.sol`, consider removing `.precision` and `.periodSize` from the `futureCashGroups` mapping and instead using getters within `FutureCash.sol` to access these values. Alternatively, consider always updating `INSTRUMENT_PRECISION` and `G_PERIOD_SIZE` within `setParameters`.

***Update:** Fixed in pull request #61. `INSTRUMENT_PRECISION` and `G_PERIOD_SIZE` are always updated in `setParameters`.*

Notes

[N01] Function visibility

For certain functions that are only called from outside the smart contract, using function visibility `public` can make the gas costs greater than using the visibility `external`. This is due to the fact that Solidity copies arguments to memory in `public` functions, whereas arguments are accessed directly in `calldata` in `external` functions. A number of functions throughout the Notional code base have `public` visibility whilst only being called externally. Some examples of this can be found below, please note this list is not exhaustive.

- `function depositIntoMarket` in `Escrow.sol`
- `function setHaircut` in `RiskFramework.sol`
- `function setContract` in `Governed.sol`
- `function initialize` in `Portfolios.sol`
- `function getRequirement` in `RiskFramework.sol`

Update: *Partially fixed in pull request #85.* `setHaircut`, `setContract`, *and* `getRequirement` *have not been changed to* `external` *however this does not impart security issues.*

[N02] Incorrect error message

In `Portfolios.sol`, line 365 checks whether 2 fCash IDs are equal. If the 2 IDs are not equal, the error code for `UNAUTHORIZED_CALLER` is returned. Consider changing this error code to more accurately reflect the error that has occurred.

Update: *Fixed in pull request #85.*

[N03] `Portfolios.sol` relies on underflow

Within the `_settleAccount` function in `Portfolios.sol`, a loop goes through each element of the `portfolio` array, occasionally removing elements from the array. When this happens, the counter is decremented immediately after so that the element now occupying the previously removed spot can be checked.

In the case that the element at index `0` is removed, this decrement will cause `i` to underflow. However, it will be incremented again at the beginning of the loop, which will set it back to `0`.

This reliance on underflow and overflow currently works, but if forced overflow checks are added to Solidity, this specific circumstance may result in reverts in later versions of the code. If forced overflow checks are eventually added to Solidity and it is desired to use that solidity version, consider implementing some other solution that does not rely on overflow/underflow. For instance, have `i` start at a value of `1`, and perform operations on the index `i-1`.

[N04] `TODO`s in code

There are “TODO” comments in the code base that should be removed and instead tracked in the project’s issues backlog. See for example line 628 of `Escrow.sol`.

Update: *Partially fixed in pull request #85, however a few “TODO” comments are still present in the code, for example line 1045 of* `FutureCash.sol`, *and line 30 of* `EscrowStorage.sol`.

defined as return values. However, there is not a `return` statement anywhere in the body of the function. Furthermore, the only spot in which this function is called makes no use of any returned values.

Consider removing the unused return parameters from the `_tradeCashPayer` function declaration.

Update: *Not fixed. Notional's statement for this issue:*

> For [N05] and [N08] these functions are removed in future versions of the code so I just ignored them

[N06] Naming issues hinder code understanding and readability

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are:

- `_updateCashLadder` to `_calculateAssetValue`.
- `settleAccount`, `settleAccountBatch` and `_settleAccount` to `settleMaturedAssets`, `settleMaturedAssetsBatch` and `_settleMaturedAssets` respectively.
- `_fetchDependencies` to `_setDependencies`.
- Consolidating or documenting the terms used to describe different assets. For example the terms `currency`, `cash`, `collateral token`, `collateral` and `local currency` are all used to refer to the currencies used for fCash, while `collateral` and `deposit currency` are both used to refer to currencies that can be used to collateralize accounts. In particular, the frequent use of the term "collateral" hinders the legibility of some sections of code.

Update: *Fixed in pull request #85, with the exception of consolidating/documenting the use of different terms to describe assets.*

[N07] Formatting abnormalities

Some inconsistencies in style and formatting were found in the codebase. For example:

such as `G_NUM_PERIODS` and `G_NUM_CURRENCIES`.

In general, consistent formatting will improve code readability and auditability, as well as making the intention clearer for any future developers. Consider fixing the identified areas to match the formatting of the rest of the codebase.

Update: *Fixed in pull request #85.*

[N08] Avoid semantic overloading of variables

Semantic-overloading in programming is the use of one variable for multiple different purposes. This can lead to code not only being harder to understand, but also harder to reason about from a security point of view.

In the function `_tradePortfolio`, and the functions it calls, `state.amountRemaining` has two different purposes depending on the situation in which `_tradePortfolio` has been executed.

When executed from `raiseCollateralViaLiquidityToken` and `raiseCollateralViaCashReceiver`, `amountRemaining` is *the amount of collateral to extract from the portfolio*. Whereas when executed from `repayCashPayer`, `amountRemaining` is *the amount of current cash available to pay off obligations*.

Consider using separate variables to store the amount of cash available to spend, and the amount of cash that needs to be raised. This will help to increase readability and understanding of your code.

Update: *Not fixed. Notional's statement for this issue:*

> For [N05] and [N08] these functions are removed in future versions of the code so I just ignored them

[N09] Misleading comments

A number of comments were found in the codebase that do not accurately represent the code.

- `FutureCash.sol` line 1074 states that `PRECISION_64x64` is a representation of `1e18`, but it is actually a representation of `1e9`.
- `Portfolios.sol` line 359 states that the fCash IDs must be the same *if the liquidation auction is not calling this function*. However the liquidation does not call this function, and it is always checked that the fCash IDs are the same.
- `RiskFramework.sol` line 115 states that "In this loop we know that the portfolio are sorted and none of them have matured". However on line 132 the code checks whether the asset has matured, and executes differently if it has.
- `FutureCash.sol` line 774 states that "This method will credit the collateralAmount back to the balances on the escrow contract", referring to the call to `_settleLiquidityToken`. However `_settleLiquidityToken` merely calculates the amount of collateral and fCash owed – it does not change any balances in FutureCash or Escrow.

Consider removing these comments or modifying them to reflect the actual code functionality. Doing so will help make the code clearer for auditors and developers.

Update: *Fixed in pull request #85.*

[N10] Typos

In the code, we found the following typos:

- `Common.sol`, line 87 should say "assets" instead of "assetd".
- `Escrow.sol`, line 22 should say "account" instead of "a account".
- `Escrow.sol`, line 523 should say "withdraw" instead of "deposit".
- `Escrow.sol`, line 669 appears to belong between lines 666 and 667, rather than where it is now.
- `Escrow.sol`, line 705 the words "to cover" can be removed.
- `Escrow.sol`, lines 741-2 the words "we have a two options here:" can be removed.
- `Escrow.sol`, line 1132 should say "appropriate" instead of "apporpriate".
- `ERC1155Token.sol`, line 204 should say "an asset" instead of "a asset".
- `FutureCash.sol` line 529 should say "exchanged" instead of "exchange"

- `Portfolios.sol`, line 211, 327, 384, 802, 898, and 923 should say “an asset” instead of “a asset”.
- `RiskFramework.sol`, line 115 should say “assets” instead of “portfolio”.

Consider fixing these typos to improve code readability and reduce confusion. Consider running codespell on future pull requests before merging.

Update: *Fixed in pull request #85.*

[N11] Unnecessary imports

The below list outlines contract imports that are unused and therefore not needed within the protocol. Consider removing these import statements to simplify the codebase and increase readability.

- Escrow.sol imports `./FutureCash.sol`
- RiskFramework.sol imports `./lib/SafeMath.sol` twice
- FutureCash.sol imports `./Portfolios.sol`
- FutureCash.sol imports `./Escrow.sol`

Update: *Partially fixed in pull request #85. The first two cases have been updated, but the latter two have not.*

Additional Recommendations

This additional section does not look to provide explicit vulnerabilities and security recommendations, but is instead provided to provoke thought and ideas around improvements to the protocol from an architectural point of view. Whilst auditing the protocol we found sections of the code complex to understand and difficult to reason about. While it is expected for decentralized finance protocols to be complex in nature, we hope that the ideas in this section might help to simplify areas of the protocol that are currently inefficient.

Pooled Over Peer-to-Peer Settling

function takes `receiver` and `payer` addresses, where the receiver has a positive balance and the payer negative, and reduces or trades the payer's assets to increase the receiver's currency balance. Such an operation is a 'peer-to-peer' operation, due to the fact that a receiver who wants to withdraw their money must find a specific payer to enable this.

If the payer has a negative balance of at least the same magnitude as the receiver, the settle can be performed in a single execution. However consider the situation where the receiver has 10,000 DAI to settle, and 5 payers each have -1234, -2817, -5127, -291 and -4012. Now the solution of who the receiver should settle against is less clear, and is going to cost the receiver 3x the amount of gas because they must settle against 3 users. This peer-to-peer settling easily leads to situations where payers, who are not incentivized to settle, end up with small negative cash balances left from previous settling transactions. Receivers who try to settle later than this will not only have the overhead of figuring out who to settle against to get their money back, but would also have to settle against many payers, maybe costing so much gas that it isn't monetarily worthwhile.

Consider instead constructing a pooled approach to settling cash balances. When a receiver wants to withdraw money from the protocol, it is known that their positive balance corresponds to negative balance that exists in the protocol – but it should not require that the balances are cancelled out at the time of withdrawal. For a simple and efficient user experience for receivers, the protocol should enable them to withdraw their currency without being required to select other peers to settle against.

Code Efficiency

As discussed in **[M08] Loops may consume too much gas**, the protocol uses a large number of loops which can be incredibly gas intensive in Solidity. For example, a single execution of `liquidate` will enter 49 loops, many of which are nested. 36 of these loops are due to the fact that a call to `freeCollateral` uses 18 loops, and this function is executed twice. This is incredibly inefficient and will quickly cause the executions to run out of gas.

Given that the majority of these loops are either `|portfolio|` or `|currencies|`, merely increasing the number of assets a user has will quickly make the cost of executing a liquidation on their account either incredibly expensive or altogether impossible.

be required to confirm whether any ideas are both efficient and secure. Consider testing the

- Storing information in mappings instead of arrays. This would remove the need for loops like those within `isDepositCurrency` and `_searchAsset` which are used to locate an item within a potentially long list.
- Combining computation into the same loop. Currently many of the loops go through a user's portfolio one asset at a time – these loops could potentially be combined to iterate through the portfolio a minimal number of times.
- Removing the need for performing a quick sort on the portfolio. During a standard liquidation, the portfolio gets sorted 3 times, executing 9 while loops. Whether by removing the need to sort altogether, or by ensuring the portfolio is always stored in order, these 9 loops could be removed.
- Separating large arrays into smaller groups of relevant information. For example, a liquidation performs two executions of `_tradePortfolio`, each of which loops through an entire portfolio looking for assets of just one type. The first iteration ignores all assets that aren't liquidity tokens, and the second ignores all assets that aren't payer tokens. If liquidity tokens and payer tokens were stored separately, this excess processing of assets that get ignored could be removed, and only relevant assets would be considered.

Conclusions

3 critical and 5 high severity issues were found during this audit. Some changes were proposed to follow best practices and reduce potential attack surface.

This first audit round has been Swapnet's initial step on its way to reach the needed level of maturity for projects intended to handle large sums of financial assets. We highly advise considering ways to refactor the protocol to be more efficient from execution, usability, and readability perspectives, and then proceed with additional rounds of security reviews only once development has concluded.

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits