KYBER NETWORK PTE. LTD.

Smart Contract & Blockchain

Code Assessment

Katalyst - Network V4

PwC Switzerland

July 7, 2020

Contents

KYBER NETWORK PTE. LTD.
8 EU TONG SEN STREET
Singapore 059818

1 Executive Summary

Dear Loi, dear Victor,

First and foremost we would like to thank KYBER NETWORK PTE. LTD. for giving us the opportunity to assess the current state of their Katalyst - Network V4 system. This document outlines the findings, limitations, and methodology of our assessment.

The projects comes with an extensive suite of tests covering all intended functionality of the smart contracts. Nevertheless some issues, for example:

- `Trading with reserve not allowed to trade this token` or
- `Removing reserves and the approval for tokens`

have been uncovered during the assessment. All issues are explained in detail in this report.

All discovered issues above with a severity higher than low have been addressed - either through code corrections or specification changes.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

Yours sincerely,

PricewaterhouseCoopers AG

Andreas Eschbach Hubert Ritzdorf

2 Assessment Overview

In this section we briefly describe the overall structure and scope of the engagement including the code commit which is referenced throughout this report.

2.1 Scope

The general scope of the assessment is set out in our engagement letter with KYBER NETWORK PTE. LTD. dated April 20, 2020. The assessment was performed on the source code files inside the Katalyst - Network V4 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	20 April 2020	dc67694f2835d33b6a0248fd2ff2b83b236e5a18	Initial Version
2	11 June 2020	f2831571466bbf5ccc43b297e54c48fbe1e08794	After First Report
3	25 June 2020	506e51fbfd4b56d18ca4260fa365502e3834156a	After Second Report
4	1 July 2020	d7a026dbcff230ec9676e8f1828762e18f24047b	After Third Report

For the solidity smart contracts, the compiler version `0.5.11` was chosen. After the intermediate report, most of the reviewed contracts were updated to solidity `0.6.6` and moved to a separate folder named `sol6`. This version of the code contained a number of changes which did not address reported findings.

We performed the following advisory services:

- Functional verification of the smart contract
- Security verification of the smart contract
- Trust verification of the smart contract
- Assessment of usage of best practice standards of smart contract designs

2.1.1 Excluded from scope

The contract files in the folders `mock`, `bridges`, `wrappers`, `utils/zeppelin`, and `emergency` are excluded from scope.

3 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview. At the end of this report section we have added subsections to describe the changes that were implemented in subsequent versions. Furthermore, in the findings section we have added a version icon to each of the findings to increase the readability of the report.

Kyber Network offers token swap services on the Ethereum Blockchain. The new edition is the fourth iteration and offers new functionality such as staking, a platform fee and a DAO.

The smart contracts can be divided into two parts, the Network and the DAO which we cover in the next two sections.

3.1 Network

The Network part of the Kyber Network system consists of the following contracts:

- `KyberNetworkProxy`, the entrypoint for the users to execute trades.
- `KybeNetwork`, contains main logic.
- `KyberStorage`, stores data about the available reserves.
- `KyberMatchingEngine`, selects the reserves to use for the token swaps based on the provided trade `hint` parameter.
- `KyberHintHandler`, contains functions to parse and build the `hint` parameter.

Some of the above contracts will communicate with other Kyber contracts. Specifically, the `KyberDAO` contract in order to retrieve the current network fee, and the `KyberFeeHandler` contract to forward trade fees to.

3.1.1 Fees

In this new version of Kyber Network reserves no longer need to pay a network fee, instead the taker needs to pay it. An optional platform fee to be paid by the taker has been introduced which allows affiliates to generate revenue when facilitating trades.

3.1.1.1 Network fee

The network fee is paid by the taker and determined by the reserve's type. However, some reserve types like utility reserves are not required to pay a network fee, Kyber never intends to charge a fee on such reserves without any capital risk. The network fee of each token swap is transferred to the `KyberFeeHandler` contract, which will distribute it three ways. This distribution is according to the BRR (burn/reward/rebate) variable which is decided upon by DAO stakers.

1) `burn`

The burn part is accumulated inside the `KyberFeeHandler` contract. Once per burn block interval the accumulated burn ether can be (partially) burned by any EOA calling `KyberFeeHandler.burnKnc()`. This function will swap ether for KNC and immediately burn the KNC.
Contracts are not allowed to call `KyberFeeHandler.burnKnc()` to prevent reserves from triggering the burning themselves and manipulate the `KNC` price. To complete this mitigation a sanity exchange rate check has been added.

2. `reward`

   The reward part is for stakers that voted in `KyberDAO` voting campaigns. When staker's vote on voting campaigns they can earn a percentage of the rewards if their voted for option is the winner.

3. `rebate`

   The rebate part is for reserves that took part in the trade. Each registered reserve has set a `rebateWallet`. Any EOA/contract can at any time call the `KyberFeeHandler.claimReserveRebate` function to withdraw the earned rebates for a specific `rebateWallet`.

3.1.1.2 Platform fee

The current version introduces a platform fee. This allows platforms like token swap aggregators to generate revenue. When initiating a trade, these can set the parameters accordingly and collect a fee up to almost 100% of the trade's value.

It is up to the users themselves to ensure the platform fee is reasonable, as the contracts do not enforce a limit.

Any EOA or contract can at any time call `KyberFeeHandler.claimPlatformFee` to make a platform wallet's acquired fees be transferred to the platformWallet address.

3.1.2 Reserves

Reserves provide liquidity to the network. In the current version, reserves need to be added by Kyber and be listed for a specific token. There are different types of reserves. Depending on the type, a network fee might or might not have to be paid during a trade. Utility reserves might have zero network fee while other types have a network fee.

3.1.2.1 Listing reserves

Anybody can deploy a reserve contract and apply for being listed in the Kyber Network. It is up to Kyber to approve a reserve and add it to the contract. Kyber will also have to separately enable the token pairs of the added reserve. The trades of listed reserves will be monitored and manually inspected (gas cost / revert rate) by Kyber. Based on these results Kyber can at any time delist reserves.

3.1.2.2 Reserve routing

The trade `hint` parameter's parsing logic has been changed compared to the previous version. The `hint` parameters is now used to indicate one of four reserve routing types to use for the token swap(s). However, this must be backwards compatible with the old `hint` parameter in order to avoid unexpected behaviour when using old hints. The four reserve routing types are:

- **Best-of-all:** The default if no hint is provided, chooses the one reserve offering the best rate.
- **Split:** Splits the trade across the provided reserves and the corresponding percentages.
- **Mask-in:** Best-of-all routing over the list of passed reserves.
- **Mask-out:** Excludes the passed reserves, performs a best-of-all routing over the remaining reserves.

3.1.2.3 Reserve rebates

Besides earnings through the exchange of tokens for ether or ether for tokens, a reserve also earns rebate for each trade it is part of. For each token swap, a part of the network fee is put aside as rebate for the reserves that took part in the token swap. This rebate is stored inside the `KyberFeeHandler` contract and can at any time, and by any EOA or contract, be transferred to the reserve's designated `rebateWallet`.

3.1.3 Token swaps

When doing a token swap in Kyber Network there are always two inner swaps performed. First the token will be swapped for ether, followed by ether being swapped for the dest token. If the swap is token to ether or ether to token one of the two inner swaps will be ether to ether.

token to token

In case of token to token the first inner swap will exchange token for ether, which is stored inside the `KyberNetwork` contract. The second inner swap will be token to ether, and will end with transferring the ether from `KyberNetwork` to the `destAddress`.

token to ether

In case of token to ether the first inner swap will exchange token for ether, which is stored inside the `KyberNetwork` contract. The second inner swap will be ether to ether, and will transfer the ether from `KyberNetwork` to the `destAddress`.

ether to token

In case of ether to token the first inner swap will be ether to ether, which will be a noop as the ether is already inside `KyberNetwork`. The second inner swap, ether to token, will transfer the ether to the reserves and receive back tokens in `KyberNetwork`. Which will then be forwarded to the `destAddress`.

3.1.3.1 Kyber Proxy contract

- `trade()`
- `tradeWithHint()`
- `tradeWithHintAndFee()`
- `swapTokenToToken()`
- `swapEtherToToken()`
- `swapTokenToEther()`

In order for users to execute trades one of the above functions in `KyberNetworkProxy` needs to be called. These functions will all end up transferring funds to `KyberNetwork`, followed by calling `KyberNetwork.tradeWithHintAndFee`, and afterwards collecting the destination amount and transferring it to the `destAddress`. Furthermore, the Kyber Proxy contract will calculate the expected trade outcome and enforce some sanity checks on the actual trade outcome, to protect the user.

In the previous versions another Kyber Proxy contract named `KyberProxyV1` was used. The new `KyberNetwork` contract was designed in such a way to be **backwards compatible** with `KyberProxyV1`.

At any time there can only be two registered Kyber Proxy contracts in the `KyberNetwork` contract.

3.1.3.2 Kyber Network contract

The `KyberNetwork.tradeWithHint` (backward compatible function for the previous proxy) and `KyberNetwork.tradeWithHintAndFee` functions can only be called by a registered Kyber Proxy. They will both initialize the trade data and call `trade()` to execute the token swap. A token swap consists of several parts, in sequence:

1. `Reserve matching`

   The `trade` function will call `calcRatesAndAmounts` to prepare both inner token swaps. This will "match" the reserves and amounts to use for both of the inner token swaps.

   The reserves and amounts are retrieved by calling the `calcDestQtyAndMatchReserves` function. This will call the `KyberMatchingEngine` to find the reserves according to the source amount, source token, dest token, and chosen reserve routing type (from the `hint`).

2. `Fee calculations`

   The `calcRatesAndAmounts` function will also calculate the platform and network fees to be paid, as well as the source to dest token rate. If a token swap is token to token the network fee will be applied twice, on the token to ether inner swap, as well as the ether to token inner swap.

3. `Max dest amount safeguard`

   After preparing all the token swap data, a check will be performed to make sure the dest amount does not exceed the `maxDestAmount` of the swap. If exceeded, the dest amount is lowered to `maxDestAmount` and the source amounts are recalculated (=lowered).

4. `Inner token swaps`

   The `doReserveTrades` function will be called for both inner token swaps. Each will will loop through the reserves for the inner token swap and execute each reserve trade. The second inner token swap will end with send the dest ether or token to the `destAddress` of the trade.

5. `Fee transfers`

   After the inner token swaps have been executed the fees are transferred by `handleFees`. This function will forward the network and platform fee to the `KyberFeeHandler` contract. In here the network fee will be further distributed between reserves and DAO stakers.

6. `GasHelper`

   Before the `trade` function completes there is an optional call to the `GasHelper` contract set inside `KyberNetwork`. This will burn gas tokens to lower the gas cost of the token transaction. For the time being this feature is disabled (contract is not set), but it can be activated in the future.

3.2 The DAO

The DAO part of the Kyber Network system consists of the following contracts:

1. `KyberDAO`, allows voting with stake on voting campaigns.
2. `KyberStaking`, allows adding/removing/delegating stake.
3. `KyberFeeHandler`, stores fees and handle fee payouts.

3.2.1 KyberStaking contract

The `KyberStaking` contract allows any account to become a staker by depositing `KNC` in the `KyberStaking` contract. A staker can at any time withdraw (part of) their stake, or delegate their entire stake to another account. Both delegating and depositing take effect from the next epoch, whereas withdrawing is immediate.

3.2.1.1 Delegating stake

Any staker can delegate his entire stake to another account, this excludes any stake delegated to this staker. An account can be set as the delegate for multiple stakers. If an account has stake delegated to it, it can place votes on behalf of those staker's stake. The Pool Master is expected to do this for a commission, and return the rest of the rewards back to the stakers. Stakers delegating their stake therefore need to claim their reward from their pool master. However, this is not part of the Kyber Network system.

3.2.1.2 Withdrawing stake

Stake withdrawal is a three step process. First, the `KyberStaking.handleWithdrawal` function will try to update the staker's stake and delegated stake accordingly. Second, the `KyberDAO.handleWithdrawal` function is called to cancel/withdraw (part of) any placed votes in still active voting campaigns. Third, the staker's latest data `KNC` balance is deducted by the to-be-withdrawn amount and the `KNC` is transferred back to the staker. If step one or two revert, the error will be caught and cause a special event to be emitted, but step three will still be executed. This to ensure a `staker` can always withdraw their stake, regardless of problems in any of the two `handleWithdrawal` calls.

When a stake gets (partially-) withdrawn this reduces the staker's reward of this epoch and a small portion of this epoch's reward will be divided over the other stakers. The reward is calculated based on the minimum that this staker had staked over the entire epoch, the voting power however is based on what he had staked during the duration of the vote only. The difference is the small portion that gets equally distributed over the other stakers.

3.2.2 KyberDAO contract

3.2.2.1 Voting campaigns

The `KyberDAO` contract lets stakers vote in voting campaigns. There are three different types of voting campaign.

1. `Network Fee`: Vote on new `network_fee` value. Can only have one Network Fee voting campaign per epoch.
2. `BRR (burn/reward/rebate)`: Vote on updating distribution of network fee (stored inside `brr_data`). Can only have one BRR voting campaign per epoch.
3. `General`: Vote on anything else.

If the `Network Fee` or the `BRR` have not been updated in an epoch, the last values remain.

To start a voting campaign, the `campaignCreator` needs to call `submitNewCampaign` providing the type, options (between 2 and 8 inclusive), begin and end timestamp, the minimum percentage of `KNC` that needs to participate in the vote, and the minimum percentage of `KNC` that the most voted for vote needs to be considered the winner. There can only be one `campaignCreator` at any time, and this will be set to an account controlled by Kyber.

3.2.2.2 *Voting*

Any EOA or contract can call `vote()`. However, vote power depends on the amount of `KNC` that is accessible to the voting account. If the voting power is zero, a vote can still be placed. However, without voting power there will be no rewards for the voter and the vote doesn't influence the vote outcome.

Stakers can place one **vote** per voting campaign. Stakers can also **update** their vote in an active voting campaign. However, to undo/cancel a placed vote in an active voting campaign a staker needs to fully withdraw his stake.

For a vote to be eligible, the staker must have staked KNC before the start of the epoch with the voting campaign, and the stake should remain staked until the end of the current epoch.

Voting power is equal to the amount of staked `KNC` by a staker (if he did not delegate) including what others delegated to this staker.

For the stakers, the reward they get is the pro-rata amount of total reward based on their voting points in this epoch.

```
(The amount of his staked KNC during the entire epoch * number of campaigns he voted for in this epoch) / totalEpochPoints
```

3.2.2.3 *Epochs*

The `KyberDAO` contract works in epochs. The time an epoch takes can only be set in the constructor and will initially be set to two weeks. During each epoch a maximum of ten voting campaigns can be created. Each voting campaign needs to start and end in the same epoch, with a minimum duration of 2 days. If there are zero voting campaigns in an epoch, there are no rewards and all received reward fees for this epoch are burnt.

3.2.2.4 *Claiming rewards*

Rewards can be claimed at any point in the future after the respective epoch ended, i.e. rewards never expire. Any EOA or contract can at any time call `claimReward`, providing a `staker` address and `epoch` number, to transfer the epoch reward of that `staker` to the `staker` address using a pull payment pattern. This function will call `KyberFeeHandler.claimStakerReward` to transfer the reward to the supplied `staker`.

3.3 Roles

3.3.1 *KyberNetwork*

`Admin`

Can withdraw any ERC20 token or ether from the KyberDao contract. Can add/remove operators/alerters, and transfer the admin role to another address. Can set the `KyberDAO`, `KyberFeeHandler`, `KyberMatchingEngine`, and `GasHelper` contracts. Can set the max gas price, and `negligibleRateDiffBps` (stored inside `KyberMatchingEngine`). Can enable the `KyberNetwork` contract. Can add/remove a Kyber Proxy contract.

`Operator`

Can add/remove a reserve, and list/delist token pairs for added reserves.

`Kyber Proxy contract`

Can call the trade functions.

`Any EOA/contract`

Can trigger the fetching of the most current network fee from `KyberDAO`, which is then stored/cached inside `KyberNetwork`. Can call any `view`/`pure` function.

3.3.2 *KyberNetworkProxy*

`Admin`

Can withdraw any ERC20 token or ether from the KyberDao contract. Can add/remove operators/alerters, and transfer the admin role to another address. Can set the `KyberNetwork` and `KyberFeeHandler` addresses.

`Any EOA/contract`

Can call the trade functions. Can call any `view`/`pure` function.

3.3.3 *KyberMatchingEngine*

`Admin`

Can withdraw any ERC20 token or ether from the KyberDao contract. Can add/remove operators/alerters, and transfer the admin role to another address. Can set the `KyberNetwork` and `KyberStorage` addresses.

`KyberNetwork contract`

Can set the `negligibleRateDiffBps`.

`Any EOA/contract`

Can call any `view`/`pure` function.

3.3.4 *KyberHintHandler*

`Any EOA/contract`

Can call any `view`/`pure` function.

3.3.5 *KyberStorage*

`Admin`

Can add/remove operators/alerters, and transfer the admin role to another address. Can set the `KyberNetwork` address. Can set the fee accounted per reserve type.

`KyberNetwork contract`

Can set the `KyberDAO`, `KyberFeeHandler` and `KyberMatchingEngine` addresses. Can add/remove a reserve, and list/unlist token pairs for an added reserve. Can add/remove a Kyber Proxy.

3.3.6 *KyberDao*

`Admin`

Can withdraw any ERC20 token or ether from the KyberDao contract. Can add/remove operators/alerters, and transfer the admin role to another address.

`Campaign Creator`

Can create a new voting campaign, cancel a not yet started voting campaign, and transfer the role to another address.

`KyberStaking contract`

Can call `handleWithdrawal` to cancel (part of) a staker's votes.

`Staker`

Can place votes using their own stake.

`Delegated Staker`

Can place votes using their own stake, and the stake of all staker's that delegated to this account.

`Any EOA/contract`

Even though possible, a voter with no stake can still call the `vote` function but it will not generate any rewards for the voter.

3.3.7 KyberStaking

`DAO contract setter`

Can one-time set the `KyberDAO` contract.

`KyberDAO contract`

Can call `initAndReturnStakerDataForCurrentEpoch` to initialize staker data inside `KyberStaking` and return it to the calling function.

`KyberStaking contract`

Can internally call `handleWithdrawal` to withdraw (part of) a staker's stake.

`Any EOA/contract`

Can deposit (=stake) `KNC`, assign a delegate, and withdraw (part of) their staked `KNC`. Can call any `view`/`pure` function.

3.3.8 KyberFeeHandler

`BurnConfig setter`

Can update the exchange rate sanity-check contract and amount of wei that can be burned in each burn interval. Can transfer the role to another address.

`DAO contract setter`

Can one-time set the `KyberDAO` contract.

`KyberNetwork contract`

Can call `handleFees` to send network/platform fees to the fee handler contract.

`KyberDAO contract`

Can call `claimStakerReward` to withdraw a staker's rewards in a certain epoch to the staker.

`Any EOA/contract`

Can call `claimReserveRebat` to transfer the rebate of a specific reserve to its rebate wallet. Can call `claimPlatformFee` to transfer the platform fee of a specific platform wallet to the platform wallet. Can call `shouldBurnEpochReward` to check if there should be no reward payout for a certain epoch and if so mark those rewards as burnable. Can call any `view`/`pure` function.

`Any EOA`

Can once per burn interval burn `KNC` which is not reserved for payout.

3.4 Trust Model

↓ Trusts →	Kyber	Reserves	Token	User	Platform
Kyber		partially	yes	no	no
Reserves	yes		yes	no	no
Token	no	no		no	no
User	no	no	yes		yes
Platform	no	no	no	no	

3.4.1 *Kyber*

Is fully trusted to:

- Add/remove reserves.
- List/unlist token pairs of reserves.
- Assign/remove the admins/operators.
- Set the references of interacting contracts.
- Not without notice replace the `KyberMatchingEngine` contract, Which could affect hint decoding/encoding and thereby a trade's matched reserves.

Needs to trust:

- The supported token contracts.
- the reserves, partially. They could inhibit trades by pulling more tokens than allowed (The trade will revert but this is a DoS attack)

3.4.2 *Reserves*

Need to trust:

- Token contracts to not unexpectedly revert on valid transfers.
- Kyber to not stop them without reason or deploy a new matching engine to their disadvantage.

3.4.3 *Token contracts*

Are fully trusted to:

- Not unexpectedly be updated such that valid transfers cause a revert.

Need to trust:

Depends on the ERC20 token contract implementation.

3.4.4 Users

Are untrusted.

Need to trust:

- Platforms, as the Kyber contracts allow a trade to have a platform fee of up to almost 100%.

3.4.5 Platforms

Are untrusted.

Need to trust:

- Kyber, must trust `Kyber` to correctly forward the platform fee to the `KyberFeeHandler` contract.

↓ Trusts →	Kyber	Staker	Poolmaster
Kyber		no	no
Staker	yes		yes
Poolmaster	yes	no	

3.4.6 Stakers

Are untrusted.

Need to trust:

- Their `Poolmaster` in case they chose to delegate.
- Kyber, to not update `KyberNetwork` to a contract not forwarding their fees.

3.4.7 Poolmaster

Are trusted by:

- Stakers having delegated to them

Need to trust:

- Kyber to not update `KyberNetwork` to a contract not forwarding their fees.

3.5 Updates in Version 2

Version 2 of the Kyber Network project introduces the following changes:

- A `KyberStorage` admin can now enable/disable rebate entitlement per reserve type.
- Several contract specific roles have been consolidated into a new `DaoOperator` role.
- `KyberFeeHandler` can accept fees in tokens instead of ether. However, this is currently not enabled and therefore fee needs to be paid in ether.

- Several sections of logic have been moved/updated.
- Several contract storage variables have moved to different contract.
- Formal trust assumptions definition was added.

3.6 Updates in Version 4

Version 4 of the Kyber Network project changes the distribution of the rebate part of the fee in case not all involved reserves are eligible for rebate:

1. Any leftover rebate is now added to the rewards instead of being available for burning.
2. If there are multiple reserves involved but only some are entitled for a rebate, rebate is only given to the entitled reserves according to `splitBps` of the trade instead of distributing the total rebate amongst the eligible reserves.

3.7 Formal trust assumptions

Directly provided by Kyber:

Network admin

- The network admin is assumed to operate the network properly and act upon the interest of KNC holders.

DAO Operator

- The DaoOperator is assumed to operate the DAO (FeeHandler, the Staking, and the DAO contract) for the best interest of the DAO which is to increase the performance of the whole Kyber Network to collect more fees.
- The DaoOperator will not change anything without a clear communication to the DAO members (aka. Staked KNC holders) and a recorded consensus between the DAO members via a general campaign
- The DaoOperator ONLY violates the second assumption IF and ONLY IF the situation requires a very fast reaction of the DaoOperator, otherwise, the whole DAO or KyberNetwork will take big damage. After such incidents, there must be clear communication to the DAO members in order to explain, discuss and implement fixes to ensure the same situation will not happen again

Takers (end users)

- A user is assumed to give enough allowance of the trading tokens to Kyber proxy contracts before the trade
- A user is assumed to be aware of the parameters, especially the platform fee which technically can be very high if they rely on malicious UI providers
- A user is assumed to set reasonable min expected rate parameter in order to protect himself from front running and market volatility
- A user should be aware that for token to Eth trades. KyberNetworkProxy expects the destination amount to be in the destination address after KyberNetwork finished processing the trade. If this amount is forwarded as part of the trade transaction the trade will revert.

Reserves

- A reserve is assumed to be aware of the fact that during a trade process, it is possible for other reserves to execute any kinds of on-chain operations in between its price lookup and actual settlement transaction.
- A reserve must be sure of the settlement price, the settlement price must be better (better for the reserves, worse for the takers) or equal to the look up price. Double checking of the settlement price is recommended and if the situation changed in a worse way for the reserves (e.g. if the reserve inventory changed in between the price lookup and settlement process), reverting the transaction is recommended.
- In case the trade's maxDestAmount is set, there is a chance the network will settle a trade with a better price (better for the reserves, worse for the takers), the reserves are REQUIRED to continue the settlement instead of reverting the transaction.
- **A reserve is assumed to take the following information around the rebate into account:**
  - If the reserve registered their inventory wallet to be rebate wallet, the rebate can be sent directly to the reserve's inventory at any time if any account claims the rebate for the reserve intentionally.
  - Dao Operator can set the rebate of a class of reserves to 0 at any time (hence it is strongly recommended not to have negative spread)
  - Rebate setting (based on the BRR) can be changed by the dao via the DAO's voting campaign, the reserves are assumed to be able to adjust their pricing accordingly
- A reserve must be sure of the settlement price, the settlement price and the price lookup are required to be consistent. Double-checking of the settlement price is recommended and if the situation changed (e.g. if the reserve inventory changed in between the price lookup and settlement process), reverting the transaction is recommended.
- A reserve doesn't front-run a taker transaction in order to make a profit and be exposed to more risk with other takers' transactions

Platforms (integrations who use Kyber to perform trades)

- A platform that uses Kyber to trade must understand the platform fee, the fee will affect the net price of end users, it increases the platform's income and reduces user experience at the same time.

The Dao (FeeHandler, Staking contract, and the DAO contract)

- The FeeHandler is assumed to be 100% online and collect fees without blocking the trades.
- The FeeHandler is assumed to receive and record platform fee information properly and enable integrated platforms to claim their platform fee later.

4 Limitations and use of report

4.1 Inherent limitations

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, PwC Switzerland has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4.2 Restriction of use and purpose of the report

Our report is prepared solely for KYBER NETWORK PTE. LTD. for use in connection with the purpose as described in the preceding paragraph. We do not, in giving our opinion, accept or assume responsibility or liability for any other purpose or to any other parties to whom our report is shown or into whose hands it may come.

5 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	**Impact**		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

6 Findings

In this section, we describe any open findings. Findings that have been resolved, have been moved to the Resolved Findings section. All of the findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3

- Incorrect Amount for Fee Calculation Risk Accepted
- Call Any payable Fallback Function / Incorrect Event Possible Code Partially Corrected Risk Accepted
- Overflows and Underflows During Calculations Risk Accepted

6.1 Incorrect Amount for Fee Calculation

Correctness Low Version 2 Risk Accepted

In case of a token-to-token trade, the network fee is computed in two parts: The fee for token-to-ETH trade and the fee for ETH-to-token trade. At the start of the ETH-to-token trade, the first fee has already been deducted:

```
uint256 actualSrcWei = tradeData.tradeWei -
    tradeData.networkFeeWei -
    tradeData.platformFeeWei;
```

The second part of the trade is computed based on the `actualSrcWei`. However, the fees of the second part are computed based on the `tradeData.tradeWei`. The fee is computed here:

```
tradeData.networkFeeWei =
    (((tradeData.tradeWei * tradeData.networkFeeBps) / BPS) * tradeData.feeAccountedBps) /
    BPS;
```

and deducted here:

```
if (src == ETH_TOKEN_ADDRESS) {
    // @notice srcAmount is after deducting fees from tradeWei
    // @notice using tradeWei to calculate fee so eth -> token symmetric to token -> eth
    srcAmountAfterFee = srcAmount -
        (srcAmount * tradeData.networkFeeBps / BPS);
```

The difference between `tradeData.tradeWei` and `actualSrcWei` will be rather small in most cases. However, we believe that the second fee should be computed based on `actualSrcWei`. Otherwise, two separate token-to-ETH and ETH-to-token trades would pay slightly less network fees (assuming that platform fees are zero) than a joint token-to-token trade. (Two separate trades would obviously consume significantly more gas. Hence, this is not an attack vector, but rather a fairness issue.)

Risk accepted:

Kyber explained that the current implementation reflects exactly what was defined in the design documentation. However, Kyber agrees there is a small difference between the two parts of a trade. Kyber states: "It is a trade off we decided to make to simplify the fee verification process to protect taker".

6.2 Call Any `payable` Fallback Function / Incorrect Event Possible

Security **Low** **Version 1** **Code Partially Corrected** **Risk Accepted**

`claimPlatformFee()` allows anyone to transfer the fees accumulated to the respective account. Note that this function features no reentrancy protection.

The transfer of the funds is done using:

```
(bool success, ) = platformWallet.call.value(amount)("");
```

1. An attacker can call any account with a non-zero `ether` amount and empty `calldata` in the name of the `KyberFeeHandler` contract. For contracts written in solidity this means any `payable` fallback function. All remaining gas will be forwarded. The problem arises as any address can collect platform fees, these fees are simply awarded to the address stored in the `struct TradeInput` of the trade. For this the attacker simply needs to execute a trade and add some balance to the desired account's earned fee tracker, `feePerPlatformWallet[address]`.

In case the `KyberFeeHandler` conract has any special permissions in other contracts, this might be misused.

2. Events could appear to be incorrect in case of reentrancy.

Consider the following example:

The Platform has a fee of 5 wei

1. Call `claimPlatformFee()` -> receive 4 wei
2. Perform a trade -> Earn some more platform fees
3. *PlatformFeePaid* emitted.

We could assume that when `PlatformFeePaid` is emitted at the end of a transaction, then the platform fee has been reset to 1, but this is not the case here.

A similar scenario might be possible with `rebateWallet` and `claimReserveRebate()`.

Code partially corrected and risk accepted:

A reentrancy check was added to `claimReserveRebate()` and `claimPlatformFee()`, this prevents the possibility of incorrect events.

6.3 Overflows and Underflows During Calculations

Design Low Version 1 Risk Accepted

While `SafeMath` is used for the relevant calculations in the `DAO` part, no `SafeMath` is used in the `Kyber Network` part. Without `SafeMath`, arithmetic operations might over- or underflow. If these are not detected, this may result in unintended behaviour of the smart contract.

The `SafeMath` library provides an efficient way to ensure the transaction would revert in case of an overflow, while only adding a little overhead in gas costs. It's use is considered part of the best practices for implementing smart contracts.

Using standard arithmetic operations as in the current implementation of the `KyberNetwork` smart contracts requires thorough checks on pre and post conditions for all operands. This increases the complexity and if not done in a structured way and when not documented how it is ensured that these arithmetic calculations are safe, makes it hard to be audited.

Several, mostly `internal`, functions in the code suffer from such over-/underflows and rely on the surrounding code to detect these and revert the transaction. If the code is updated, the usage of these functions might be different and a check that previously prevented a transaction with an overflow to successfully complete may not be sufficient anymore.

`KyberNetwork.getExpectedRateWithHintAndFee`, a function that allows users to query the expected rate for a trade can be called successfully with parameter `platformFeeBps` set to a carefully chosen value so that one or multiple calculations in `calcRatesAndAmounts()` overflow. Despite being a `view` function it should never return a wrong result as otherwise implementors have to put their own safeguards in place every time they call this function.

Risk accepted:

The mentioned overflow in `KyberNetwork.getExpectedRateWithHintAndFee()` is now prevented, a check ensures that `platformFeeBps` doesn't exceed the value of `BPS`. An additional possiblity where an overflow might occur has been mitigated inside `calcSrcAmountsAndGetRates()`.

Kyber explained that their strategy for detecting overflows and underflows is to *review all arithmetic operations and condition checks and look for possible boundary conditions*.

7 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	2

- Trading With Reserve Not Allowed to Trade This Token Code Corrected
- totalEpochPoints Not Matching Specification Specification Changed

Medium-Severity Findings	3

- Amount for Rate Query Might Be Higher Than Actual Trade Amount Specification Changed
- Removing Reserves and the Approval for Tokens Code Corrected
- The First Bytes of reserveID Might Not Match reserveType[reserveID] Specification Changed

Low-Severity Findings	15

- Inconsistent bytes32 Comparison Code Corrected
- Superfluous setDecimals Call Code Corrected
- Unnecessary Event Parameter bool add Code Corrected
- Unused Field rateWithNetworkFee of Struct TradeData Code Corrected
- Unused Import IKyberFeeHandler.sol in KyberDao.sol Code Corrected
- Duplicate View Function for negligibleRateDiffBps Code Corrected
- Inconsistent Delegated Event Code Corrected
- Redundant Array delete Code Corrected
- Redundant Assertions Code Corrected
- Reward/rebate/BPS Off-By-One Code Corrected
- Unnecessary Check endTimestamp == 0 Code Corrected
- Use assert for "Should Not Happen" Code Corrected
- latestNetworkFeeResult Initial Value Hardcoded but Overwritten in Constructor Code Corrected
- require(validateVoteOption) Never Fails Code Corrected
- validateCampaignParams of KyberDAO Allows Incompatible startTimestamp and startEpoch Code Corrected

7.1 Trading With Reserve Not Allowed to Trade This Token

Design High Version 1 Code Corrected

The `KyberNetwork` contract implements `listPairForReserve()`. This function is annotated with `Allow or prevent a specific reserve to trade a pair of tokens` and can only be called by the operator. This pushes or removes the reserve id to/from `reservesPerTokenDest` or `reservesPerTokenSrc` in the `KyberStorage` contract.

During a trade however these limitations only have an effect for a `best-of-all` trade if no hint is provided, or in case the hint is of type `Mask-out`. Only in these cases the available reserves are retrieved from the `KyberStorage` contract.

In case of hints for either `Mask-in` or `Split` this limitation is not enforced and if specified and only if this reserve returns a non-zero rate, the trade continues to execute normally using the reserves passed in the hint. There are no checks if these reserves are allowed to trade this token pair.

The design of `KyberNetwork` requires the reserves to pull the tokens using `transferFrom()` themselves from `KyberNetwork`, ether however is transferred automatically with the call. `listPairForReserve` approves the reserve address which is required for them to transfer the tokens successfully. Consequently, for legitimate tokens, the scenario above only works for ether to token trades.

In case of a malicious reserve, this opens up a new vector of possible attacks. The only protection against attempting to trade ether for an unsupported token is that the rate of the exchange after fees must be non-zero. This is checked inside `trade()` of `KyberNetwork`. Now a malicious reserve can return an arbitrary rate for an arbitrary (token-) address circumventing this check. The malicious token may also simply allow `transferFrom()` as required for the transfer to work. The implications of this need to be investigated.

Code corrected:

The reserve IDs are now validated. This is done as follows: `KyberStorage.getReservesData()` has been enhanced to check this. It returns an additional boolean `areAllReservesListed` which is `true` only if all involved reserves are listed to trade this pair. During a trade, `KyberNetwork.calcDestQtyAndMatchReserves()` retrieves the data of the involved reserves and will revert if the returned value for `areAllReservesListed` is not `true`.

7.2 `totalEpochPoints` Not Matching Specification

Correctness High Version 1 Specification Changed

The specification document states:

```
totalEpochPoints (uint -> uint): total points for an epoch (total votes f
or all campaigns in that epoch)
```

This does not hold in the current implementation, it does not necessarily represent the total votes for all campaigns in that epoch.

`handleWithdrawal()` of the `KyberDAO` contract is responsible for correctly deducting votes in case of a withdrawal.

If the `staker` has voted in the current epoch, this is done as follows:

```
totalEpochPoints[curEpoch] = totalEpochPoints[curEpoch].sub(numVotes.mul(reduceAmount));

// update voted count for each campaign staker has voted
uint256[] memory campaignIDs = epochCampaigns[curEpoch];

for (uint256 i = 0; i < campaignIDs.length; i++) {
    uint256 campaignID = campaignIDs[i];

    uint256 votedOption = stakerVotedOption[staker][campaignID];

    if (votedOption == 0){
       continue;
    } // staker has not voted yet

    Campaign storage campaign = campaignData[campaignID];

    // deduce vote count for current running campaign that this staker has voted
    if (campaign.endTimestamp >= now) {
        // user already voted for this campaign and the campaign is not ended
        campaign.campaignVoteData.totalVotes = campaign.campaignVoteData.totalVotes.sub(reduceAmount);
        campaign.campaignVoteData.votePerOption[votedOption - 1] = campaign
            .campaignVoteData
            .votePerOption[votedOption - 1]
            .sub(reduceAmount);
    }
}
```

Initially, the value of `totalEpochPoints[curEpoch]` is reduced by the amount of `numVotes * reduceAmount`. The total votes, however, are only deducted if the campaign has not ended yet `if (campaign.endTimestamp >= now)`.

This finding leads to a discrepancy between `totalEpochPoints` and its meaning: `total votes for all campaigns in that epoch`.

Specification changed:

This variable represents the total amount of eligible votes in an epoch. When a staker withdraws, a vote will not be affected and remain eligible if the campaing has already ended.

The specification was updated and now states: *total points for an epoch which will be used to determine reward portion(in percentage) for KNC stakers. The total point is deducted when a staker withdraws his KNC.*

7.3 Amount for Rate Query Might Be Higher Than Actual Trade Amount

Design | Medium | Version 2 | Specification Changed

In case the calculated destination amount of the trade exceeds the maximum destination amount specified, the required source amount to reach the target amount is recalculated and the change is refunded. `calcTradeSrcAmountFromDest()` is used to recalculate and update the trade data. This function uses the previously determined reserves and rates.

Hence when the trade is executed the actual reserve trade (or trades in case of `trade.split`) may happen using a lower amount then the amount used to query the rate of the reserve.

A reserve might expect the trade to be exactly of the amount previoulsy used to query the rate. During the initial part of the audit we asked to clarify the assumptions about the reserves, the updated specification contains the following sentence:

```
A reserve must be sure of the settlement price, the settlement price and
the price lookup are required to be consistent.
```

If reserves revert the transaction in this case, all trades where the `maxDestAmount` has been exceeded and the source amount had to be recalculated will fail which would break this functionality entirely.

Querying a rate of a different amount might result in a different rate, better or worse. This can lead to unexpected behaviour. Reserves have to be very careful when detecting a missmatch between the amount used for querying the rate and the amount of the actual trade.

Specification changed:

The specification now includes a section on reserve's expected behaviour in case of max dest amount exceeded. In such cases the reserve should make sure that the new recalculated trade is "better for the reserve, worse for the taker".

7.4 Removing Reserves and the Approval for Tokens

Design Medium Version 1 Code Corrected

After a reserve has been added using `KyberNetwork.addReserve()` it needs to be listed to trade token pairs. This is done using `KyberNetwork.listPairForReserve()`. This function allows a specific reserve to trade a pair of tokens and in case of `tokenToEth` approves this reserve to transfer tokens from `KyberNetwork`. It can also be used to remove the allowance.

A reserve can be removed. To update the reserve address for a `reserveID`, the reserve needs to be removed before the new address can be added. Neither the approval for ERC-20 token transfer is reset nor the entries in the mappings `reservesPerTokenSrc` or `reservesPerTokenDest` of `KyberStorages` are removed.

Note that this must be done manually using `KyberNetwork.listPairForReserve()` before the reserve is removed as afterwards this is not possible anymore.

This has two consequences:

1. If the allowance is not reset and the `KyberNetwork` contract happens to have a non-zero balance of these tokens, such a removed reserve could still transfer these tokens.
2. Entries in the mappings `reservesPerTokenSrc` and `reservesPerTokenDest` of `KyberStorages` would persist. Even without `KyberNetwork.listPairForReserve()` being called for the new reserve address, the `KyberMatchingEngine` would continue to use

this reserve as possible candidate for such a token swap. If this reserve is selected for a token to ether trade, this trade would fail as the approval required for transferring the token using `transferFrom()` is missing.

Code corrected:

In the updated implementation the functionality to add/remove reserves has been removed from the `KybeNetwork` contract and this is now entirely handled inside the `KyberStorage` contract. `removeReserve()` now ensures all tokens have been delisted, which resets the approval to 0, before removing the reserve. In case a reserve has too many tokens and the removal requires too much gas, the network has to delist those tokens manually and remove the reserve afterwards.

7.5 The First Bytes of `reserveID` Might Not Match `reserveType[reserveID]`

Correctness Medium Version 1 Specification Changed

The specification document `Katalyst - Network V4 changes` on page 3 describes the different type of reserves and their respective reserve ID as follows:

The reserve ID is a 32 bytes where its first byte determines the type of the reserve, the other bytes will hold a unique name per reserve. The table below lists all of the reserve types and their first byte

This does not hold in the current implementation. The `reserve type` of a `reserve` is determined by the value stored in the mapping `reserveType` of the `KyberStorage` contract. To determine the reserve type, the current code relies on the value stored in this mapping.

A reserve can be added by Kyber using the `addReserve` function of the `KyberNetwork` contract. The reserve id and the reserve type are passed as separate parameters. This calls `addReserve()` of the `KyberStorage` contract and adds the reseve. There is no check if the first byte of the reserve id matches the reserve type and hence an "invalid" entry might be created.

Specification Changed:

This requirement is outdated and has been removed from the specification.

7.6 Inconsistent `bytes32` Comparison

Design Low Version 2 Code Corrected

The `KyberStorage.addReserve` function contains the following two lines:

```
require(reserveAddressToId[reserve] == bytes32(0), "reserve has id");
require(reserveId != 0, "reserveId = 0");
```

reserveAddressToId[reserve] and reserveId are both of type bytes32. Only in the first require statment the the zero is explicitly casted to a bytes32. The second check could be enhanced to be more explict.

Code corrected:

The second check was updated to also cast 0 to bytes32.

7.7 Superfluous setDecimals Call

Design Low Version 2 Code Corrected

The KyberNetwork.listTokenForReserve function calls setDecimals both when adding and removing a reserve's approval.

```
if (add) {
   token.safeApprove(reserve, 2**255);
} else {
    token.safeApprove(reserve, 0);
}
setDecimals(token);
```

If a token is added, decimals is fetched from the token contract and stored inside KyberNetwork. If a token is removed the decimals do not need to be fetched and stored again.

Code corrected:

The setDecimals call was moved into the first if body, only executing when adding a reserve token pair listing.

7.8 Unnecessary Event Parameter bool add

Design Low Version 2 Code Corrected

The AddReserveToStorage event inside KyberStorage has a bool add parameter. In the previous version of the audited code the add parameter was used to indicate adding or removal of a reserve. In the current version removing a reserve has its own event RemoveReserveFromStorage. Also, the bool add parameter is always set to true in emitted AddReserveToStorage events. Therefore, the bool add parameter could be removed, and it would still be possible to track adding vs removal of reserves.

Code corrected:

The bool add event argument was removed.

7.9 Unused Field `rateWithNetworkFee` of Struct `TradeData`

Design Low Version 2 Code Corrected

`KyberNetwork` defines the struct `TradeData`:

```
struct TradeData {
    TradeInput input;
    ReservesData tokenToEth;
    ReservesData ethToToken;
    uint256 tradeWei;
    uint256 networkFeeWei;
    uint256 platformFeeWei;
    uint256 networkFeeBps;
    uint256 numEntitledRebateReserves;
    uint256 feeAccountedBps;
    uint256 entitledRebateBps;
    uint256 rateWithNetworkFee;
}
```

The field `rateWithNetworkFee` is never set nor read. `KyberNetwork.trade()` has an internal variable `uint256 rateWithNetworkFee` which is used to temporarily store the returned value of `calcRatesAndAmounts`.

Code corrected:

The unused field `rateWithNetworkFee` was removed.

7.10 Unused Import `IKyberFeeHandler.sol` in `KyberDao.sol`

Design Low Version 2 Code Corrected

In previous iterations of the implementation, stakers had to claim their reward in the `KyberDao` contract which called the respective function of the `KyberFeeHandler` contract. This has been changed and stakers now directly claim their reward from `KyberFeeHandler`.

The current implementation of `KyberDao` still imports the `IKyberFeeHandler` interface which is no longer used by this contract.

Code corrected:

The unused import was removed.

7.11 Duplicate View Function for `negligibleRateDiffBps`

Design | Low | Version 1 | Code Corrected

The `negligibleRateDiffBps` variable inside `KyberMatchingEngine` is declared as `public`. Therefore, an automatic getter function called `negligibleRateDiffBps` is created. However, there is also a function called `getNegligibleRateDiffBps`, which simply returns `negligibleRateDiffBps`.

Code corrected:

The `negligibleRateDiffBps` variable is no longer declared as `public`, thus defaulting to `internal`.

7.12 Inconsistent `Delegated` Event

Design | Low | Version 1 | Code Corrected

The event

```
event Delegated(
    address indexed staker,
    address indexed delegatedAddress,
    uint256 indexed epoch,
    bool isDelegated
);
```

is emitted whenever the state of a delegated stake changes. Removing a previously set delegation is only possible by delegating to oneself which leads to spurious event emissions. The actual storage changes differ depending on the situation, this inconsistent behaviour may cause confusion for dApps processing these events.

The `delegate` function of the `KyberStaking` contract allows one to delegate ones stake to any address except to `0x0`. Delegating to oneself is equal to undoing a previously set delegation.

In general, if the stake was previously delegated, one `Delegated` event with the field `isDelegated` set to `false` will be emitted, indicating this delegation is ending. Such an event is always followed by another `Delegated` event with the field `isDelegated` set to `true` and the new delegated address. This also holds true for the case when one delegates to one's own address.

While the delegated address is actually set, the amounts for both `delegatedStake` entries are not set in this case as one cannot delegate to oneself - the event should not be emitted as no actual delegation took place. One event with `isDelegated` set to `false` is sufficient to indicate that a delegation has been withdrawn.

Code corrected:

The mentioned `Delegated` event is now only emitted if the new delegate is not the staker.

7.13 Redundant Array `delete`

Design | Low | Version 1 | Code Corrected

The `KyberDAO.cancelCampaign` function removes a campagin id from the `campaignIDs` array by doing:

```
delete campaignIDs[campaignIDs.length - 1];
campaignIDs.pop();
```

The `delete` is unnecessary as pop will also delete the last element, before decreasing the array length by 1.

Code corrected:

The `delete` was removed.

7.14 Redundant Assertions

Design | Low | Version 1 | Code Corrected

The `delegate()` function of the `KyberStaking` contract features redundant assertions.

```
assert(stakerPerEpochData[curEpoch + 1][curDAddr].delegatedStake >= updatedStake);
assert(stakerLatestData[curDAddr].delegatedStake >= updatedStake);

stakerPerEpochData[curEpoch + 1][curDAddr].delegatedStake =
    stakerPerEpochData[curEpoch + 1][curDAddr].delegatedStake.sub(updatedStake);
stakerLatestData[curDAddr].delegatedStake =
    stakerLatestData[curDAddr].delegatedStake.sub(updatedStake);
```

The SafeMath `sub()` would revert in case of underflow, making the previous assertions redundant.

Code corrected:

The two assertions have been removed.

7.15 Reward/rebate/BPS Off-By-One

Correctness | Low | Version 1 | Code Corrected

The `KyberDao.validateCampaignParams` contains the following check:

```
require(
    rewardInBps.add(rebateInBps) <= BPS,
    "validateParams: rebate + reward must be smaller then BPS"
);
```

This will not cause a revert if reward + rebate equals BPS. So either the code or the comment is incorrect.

Code corrected:

The comment was updated to reflect the code.

7.16 Unnecessary Check `endTimestamp == 0`

Design | Low | Version 1 | Code Corrected

The `KyberDAO.getCampaignWinningOptionAndValue` function starts by performing the following checks:

```
Campaign storage campaign = campaignData[campaignID];
if (!campaign.campaignExists) {
    return (0, 0);
} // not exist

// not found or not ended yet, return 0 as winning option
if (campaign.endTimestamp == 0 || campaign.endTimestamp > now) {
    return (0, 0);
}
```

However, the `campaign.endTimestamp == 0` check is unnecessary as the `!campaign.campaignExists` check will also trigger when `endTimestamp == 0`.

Code corrected:

The redundant check `campaign.endTimestamp == 0` has been removed.

7.17 Use `assert` for "Should Not Happen"

Design | Low | Version 1 | Code Corrected

The `KyberDAO.getStakerRewardPercentageInPrecision` function contains the following code:

```
// something is wrong here, points should never be greater than total pts
if (points > totalPts) {
    return 0;
}
```

If this should not happen, making it an `assert` might be more suitable.

Code corrected:

The `if` was replaced with an `assert`.

7.18 `latestNetworkFeeResult` Initial Value Hardcoded but Overwritten in Constructor

Design Low Version 1 Code Corrected

In the `KyberDao` contract, the variable `latestNetworkFeeResult` is declared and set to 25.

```
uint256 internal latestNetworkFeeResult = 25; // 0.25%
```

Inside the constructor however, this is immediately overwritten using the value passed as argument `_defaultNetworkFeeBps`:

```
latestNetworkFeeResult = _defaultNetworkFeeBps;
```

The hardcoded value might be misleading for inexperienced readers looking at the smart contract.

Code corrected:

The hardcoded value was removed.

7.19 `require(validateVoteOption)` Never Fails

Design Low Version 1 Code Corrected

The `KyberDAO.vote` function validates the voted for option for the campaign by doing:

```
require(validateVoteOption(campaignID, option), "vote: invalid campaignID or option");
```

However, the `validateVoteOption` itself will either return true or revert. Therefore, the `require` around `validateVoteOption` inside `vote()` will never fail. Also, returning `true` from `KyberDAO.validateVoteOption` is unnecessary.

This pattern of unnecessarily returning true from a function and consequently wrapping it in a require is present in multiple places in the smart contracts.

Code corrected:

The `validateVoteOption` function, as well as several other functions, no longer `return true` or are called wrapped in a `require`.

7.20 `validateCampaignParams` of `KyberDAO` Allows Incompatible `startTimestamp` and `startEpoch`

Correctness **Low** **Version 1** **Code Corrected**

This function validates parameters for a new campaign and returns true only if the parameters are valid to start a new campaign. The parameters of this function include `startTimestamp` and `startEpoch` however no check ensures these values are compatible.

A `startTimestamp` in the `current epoch`, `startEpoch` set to `current epoch + 1` and an `endTimestamp` in `current epoch + 1` might result in this function returning true.

Note that the `submitNewCampaign` function that calls this `validateCampaignParams` function is not affected as the passed epoch matches the timestamp.

Code corrected:

The function parameter `startEpoch` of `validateCampaignParams()` has been removed and the epoch is now calculated based on `startTimestamp`. This resolves the issue described above.

8 Notes

8.1 Confusing Comment

Note **Version 1**

The `KyberMatchingEngine.doMatch` function contains the following code and comment:

```
// if this reserve pays fee its actual rate is less. so smallestRelevantRate is smaller.
bestReserve.destAmount = (bestReserve.destAmount * BPS) / (BPS + negligibleRateDiffBps);
```

The code calculats the acceptable `destAmount` considering the `negligibleRateDiffBps`. This is then used to craft a lis of all reserves offering a similar rate within the acceptable negigible difference. This is irrespective of wether the reserve has to pay a fee or not and is only based on the resulting `destAmount`.

While the comment itself is not wrong, it does not describe what happens on the following line and might be confusing.

8.2 Front-running `KNC` Burning

Note **Version 1**

KNC burns occur at predictable times and can be triggered by anyone. As they involve the purchase of `KNC` tokens, they can be front-run. However, Kyber has taken a number of meaningful steps to limit potential benefits of front-running:

- The rate is compared to a secondary source and cannot differ by more than 10%
- At most 2 `ETH` can be transferred into `KNC` and burnt
- The burning can only be triggered by an externally owned account

Therefore, under ideal conditions the attacker can gain at most 0.2 `ETH`. Hence, we perceive this to be safe unless:

- The secondary source provides incorrect data
- There is a significant price change and the secondary source is not up-to-date
- The value of 0.2 `ETH` raises immensely

These conditions are out of the control of Kyber. Hence, no change is required from our side. In the worst case the consequence for Katalyst - Network V4 will be that less `KNC` are burnt than expected.

8.3 Imprecise Specification

Note **Version 1**

We detected some discrepancies between the specifications provided and the actual implementation. The following findings are nothing major, thus they are summarized in this issue:

1. For `Kyber DAO specifications (iteration 2).pdf`

`voting power is equal to the eligible stake of the staker at each epoch.`

This however is not accurate. A staker can have different voting power for votes in different campaigns during one epoch. Consider the following scenario:

Initial a staker has 100 `KNCs` staked, votes for campaign A option 1. After this campaign ended, he withdraws 50 `KNCs` and votes for campaign B option 1.

- For campaign A his voting power was 100 `KNCs`
- For campaign B his voting power was 50 `KNCs`

The actual voting power is equal to the eligible stake of the staker at the `endTimestamp` of the vote.

2. For `Kyber Staking specifications.pdf`

On page one the phraseology `voting power at each epoch is used` which is imprecise (see above).

This document also states `address(0)` means the staker has not delegated to anyone. This is incorrect, `address(0)` is only the case before this `stakerPerEpochData` mapping entry has been initialized, afterwards, if undelegated `delegatedAddress` is equal to the `staker`.

8.4 Inconsistent Handling to Ensure Non-Zero Address

Note Version 1

The `KyberStorage` contract ensures that the passed arguments are non-zero.

```
require(_feeHandler != IKyberFeeHandler(0), "feeHandler 0");
require(_matchingEngine != address(0), "matchingEngine 0");
IKyberMatchingEngine newMatchingEngine = IKyberMatchingEngine(_matchingEngine);
```

The handling is inconsistent and differs by contract. For better readability this could be simplified.

8.5 Indexing `link` of `NewCampaignCreated`

Note Version 1

`KyberDao` defines the following event:

```
event NewCampaignCreated(
    CampaignType campaignType,
    uint256 indexed campaignID,
    uint256 startTimestamp,
    uint256 endTimestamp,
    uint256 minPercentageInPrecision,
    uint256 cInPrecision,
    uint256 tInPrecision,
    uint256[] options,
    bytes link
);
```

The parameter `bytes link` could be indexed. This would allow filtering the events for a given link and retrieve the according campaign information easily.

8.6 Minimum of One Interaction With `KyberNetwork` in Every Epoch

Note **Version 1**

For the purpose of this code review we assume that at least one trade or query for an exchange rate happens during each epoch. The duration of an epoch will be set to 2 weeks. This is important to ensure the `network fee` values are properly updated using `getAndUpdateNetworkFee()`.

Otherwise following scenario is possible:

- A successful voting campaign on new parameters for the `network fee` takes place in `epoch x`
- There is no interaction with `KyberNetwork` in epoch `x+1`, consequently the `network fee` values are not updated
- No voting campaign on the `network fee` takes place in epoch `x+1` as this fee has just recently been updated
- Trades in epoch `x+2` will reuse the old outdated `network fee` from before as the update from the voting campaign in epoch `x` has been missed by `KyberNetwork` in epoch `x+1`

8.7 Outdated Comments About First Byte of the Reserve ID

Note **Version 2**

Previsouly the first byte of the reserve ID designated the reserve's type. While this requirement has been removed from the specification document, this can still be found as comment throughout the codebase:

`KyberStorage.addReserve()` on line 129: */// @param reserveId The reserve ID in 32 bytes. 1st byte is reserve type*

`KyberStorage.getReserveDetailsByAddress()` on line 503: */// @return reserveId The reserve ID in 32 bytes. 1st byte is reserve type*

`KyberMatchingEngine.getTradingREserves()` on line 62: */// @return reserveIds Array of reserve IDs for the trade, each being 32 bytes. 1st byte is reserve type*

8.8 Platform Fee up to Almost 100%

Note **Version 1**

While the documentation clearly states:

```
The platform calling the trade can define any platform_fee up to almost 1
00% of the trade value (for the sake of simplicity, we decided not to for
ce any strict upper bound for this param).
```

This creates a rather high risk for the users. Users trading via an affiliate might not even know that Kyber is used for their trade or that they need to be careful. If something goes wrong, for the unaware user it might look like Kyber collected this fee. A simple sanity check would reduce this trust issue.

8.9 Tokens With Transfer Fees

Note **Version 1**

As in previous implementations of Katalyst - Network V4 special care must be taken regarding tokens that have transfer fees. These tokens cause issues as their transferred amounts into and out of the `KyberNetworkContract` differ due to the fees that were deducted. Hence, the consistency checks for the transferred amounts will fail and the transaction will, correctly, revert.