

Expert Security Audit

of Kyber Network v2

June 29, 2018

Produced for



KYBER
NETWORK

by



CHAINSECURITY

Table Of Content

Foreword	1
Executive Summary	1
System Overview	2
Order Book Matching	2
Trust Model	4
User Perspective	4
Reserve Perspective	4
Network Perspective	4
Audit Overview	6
Scope of the Audit	6
Depth of Audit	6
Terminology	7
Scope	8
Included in the scope	8
Limitations	10
Details of the Findings	11
Security Issues	11
1.1 Kyber Reserve ✓ No Issue	11
1.2 Volume Imbalance ✓ No Issue	12
1.3 User requirements ✓ No Issue	12
1.4 Network requirements (partial) ✓ No Issue	13
1.5 Overflow in reserveFeeToBurn possible L	14
1.6 Overflow in getImbalance possible L ✓ Acknowledged	14
1.7 Influence of maxDestAmount on chosen conversion rate M ✓ Acknowledged	14

1.8	Failing transaction due to rounding issue	M	✓ Acknowledged	15
1.9	Locked tokens or ETH	✓ No Issue		16
Trust Issues				16
2.1	Wrong require in setMinSlippageFactor can lead to Underflow	M	✓ Fixed	16
2.2	Additional Proxy Checks possible	L	✓ Fixed	16
Design Issues				17
3.1	Capitalization of constants	L		17
3.2	Specification mismatch	L	✓ Addressed	17
3.3	validateTradeInput should fail by default	M	✓ Fixed	17
3.4	Lint warnings disabled with no explanation	M	✓ Addressed	17
3.5	transferAdminQuickly bypasses sanity checks	M	✓ Acknowledged	17
3.6	Code consistency	L		18
3.7	Inconsistent use of feeAmount in sendFeeToWallet	L		18
3.8	Lack of indexed fields in some events	L	✓ Acknowledged	18
3.9	Off-by-one error with MAX_QTY	L	✓ Fixed	19
3.10	Assigning to function parameters	L	✓ Acknowledged	19
3.11	Gas savings possible	L	✓ Acknowledged	19
3.12	Front-running possible	L	✓ Acknowledged	19
Recommendations / Suggestions				20
Disclaimer				22

Foreword

We first and foremost thank KYBER.NETWORK for giving us the opportunity to audit their smart contracts. This document outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

CHAINSECURITY is overall convinced of the soundness of KYBER.NETWORK's project with regards to its design and its smart contract implementations. The smart contract test suite is exhaustive, and the smart contract code is of high quality. During the audit, CHAINSECURITY uncovered several issues worthy of KYBER.NETWORK's attention, which have mostly been addressed. Overall, no significant security concern remains.

System Overview

The KYBER.NETWORK exchange consists of two main conceptual components:

- The network manages the platform, including fees, trading and reserves.
- The reserves hold funds and exchange them with market participants.

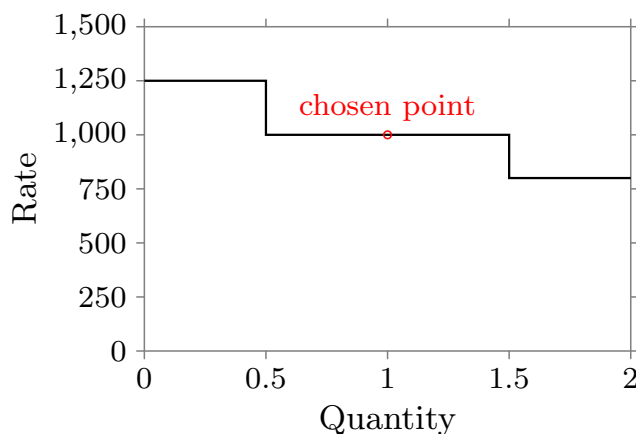
Both components consist of several separate smart contracts, whose architecture and trust model are described in detail in a technical blogpost by KYBER.NETWORK¹.

Order Book Matching

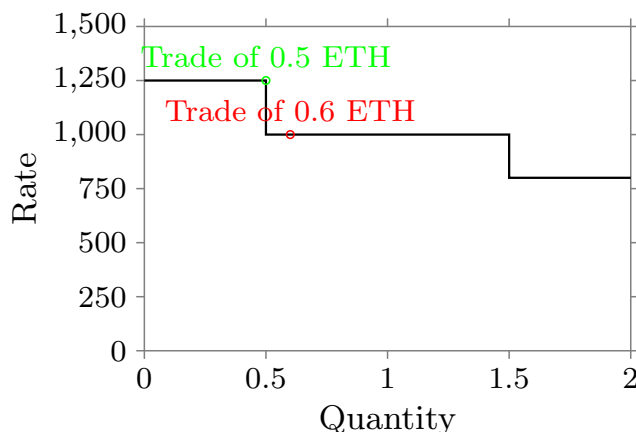
There are a few (potentially unexpected) details about the order book matching. These, however, have valid reasons from CHAINSECURITY's point of view.

No partial completion: An order is never partially completed: It is either totally completed or not at all.

Same rate and reserve for complete order: If an order is completed, all of the funds are provided by the same reserve at the same rate. Even though the order book of this reserve can contain different rates. Consider the example below:



In this example all funds will be converted using a rate of 1000 even though some funds are available at a better rate of 1250. This has the consequence that paying more can lead to a lower payout. For such a case, consider the example below:



¹<https://blog.kyber.network/kyber-network-smart-contract-29fd7b46a7af>

The shown trade of 0.6 ETH results in a rate of 1000 and thereby 600 KNCs. The shown trade of 0.5 ETH results in a rate of 1250 and thereby 625 KNCs. Hence, larger trades can potentially result in a smaller output.

Note, that such issues can be partially resolved on the application layer by checking the order book structure and, if needed, making multiple trades. This is only a partial resolve due to the inherent race condition of checking something inside the blockchain state and then acting upon it.

Trust Model

In this section we briefly describe the trust model for the different actors inside the KYBER.NETWORK.

User Perspective

Users of the KYBER.NETWORK want to exchange funds, but do not want to put their invested funds at risk. This security property collides with KYBER.NETWORK's requirement for updatable code, but KYBER.NETWORK has found a good compromise:

A user's funds are protected by the `KyberNetworkProxy` which forwards the request to the current implementation but checks that the execution was according to the client-specified requirements. In particular, it checks that the minimum exchange rate was upheld. Note that no guarantee for the best possible exchange rate is made.

In order to protect the minimum exchange rate, the user has to make sure that it has claimed all available funds of the target token. Some services or tokens allow anyone to trigger a payout on behalf of a beneficiary. If such payouts are available for the destination address the reserve can simply trigger the payout instead of sending funds.

Just because users are protected from the effects of misbehaving reserves, they still cannot trust them. In particular, users shouldn't trust reserve-emitted events as these can be manipulated.

However, the user is not entirely free from risk. A malicious network or reserve can steal the user's gas. Additionally, whenever the user trades malicious tokens (tokens with insecure contracts) all funds involved in this transaction can be lost.

Users must review

If they want to independently protect their funds, users of KYBER.NETWORK need to review:

Proxy To ensure that it protects their interests and performs the necessary checks.

Traded Token To ensure that the token is not malicious, as the user can otherwise lose their tokens or they can become worthless.

Reserve Perspective

A reserve aims to offer exchange services through the KYBER.NETWORK. However, KYBER.NETWORK administrators should have no control over the funds of a reserve.

The reserves pay for the participation inside KYBER.NETWORK in KNCs. This payment can be manipulated by KYBER.NETWORK, because KYBER.NETWORK controls the exchange rate. Therefore, reserves can limit their exposure and only make KNCs available once they agree with the exchange rate.

In the current contracts, the reserves fully trust the network contract, which is fine as long as these contracts remain well-behaved. Therefore, reserves have to be very careful to update to future versions of the network contract.

Reserves must review

If they want to independently protect their funds reserves of KYBER.NETWORK need to review:

Network Contract This check has to be renewed for each update to the network contract and has to consider the complex relation between the two.

Offered Tokens To ensure that funds in these tokens cannot be suddenly lost.

Network Perspective

The network administrators need to perform proper KYC operations on the reserves. This is to ensure a possible punishment in case of misbehavior. The network needs to collect KNCs from the reserves as payment. These token transfers currently require cooperation from the reserves.

Network should review

The network's resources are generally not at risk. However, in order to ensure a good customer experience, the network should review the following items:

Honest, Efficient Reserves To avoid undesirable performance, the network should ensure that reserve contracts are gas-efficient, don't steal and of user gas and act as proper reserves. A single misbehaving reserve can block the network.

Proper Tokens In order to ensure disappointments when exchanging, KYBER.NETWORK should check the basic properties for all listed tokens.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on June 6, 2018, and updated on June 27, 2018²:

File	SHA-256 checksum
ConversionRates.sol	12caf9522e2c5153255e6bc62a3560f4f63ff20eb97c00b312ad0c8ed1ac1d38
ConversionRatesInterface.sol	bac394975da9ba41c0c1130895a25336c98e2341102c6f8fa7494ff03d94f0bc
ERC20Interface.sol	b6c83117f2507859921f6abcc208b5a52d4a08169e259d734b518bc304950c1c
ExpectedRate.sol	3706894b63a2dd5ff1726c955f59a575cf616ae1631602190f8f633cb17e9fe2
ExpectedRateInterface.sol	3ae93286e8713098d3f44c0915a43b9e64f3acaf490b3d60fed4e2791e3ab2c4
FeeBurner.sol	9f99aa3f69f19d8c6e2a970a3bd55b6095de7c94719f12ff9ab5547f184027bc
FeeBurnerInterface.sol	b5e2f31806b34b5402b38b934c79234a2a0b480619f4b2961772ca8e0d9446cb
KyberNetwork.sol	b4f931a7d58445929897c4bc4a42d1542af2a5b59101f7ea3af710c7c49e3996
KyberNetworkInterface.sol	c934f99e45194407148dcfc6224ea132eaa8d4b3c8d98ff2aaf935defadec944
KyberNetworkProxy.sol	7320bd6ae2f310c5d3d2f74e5e0d888be03c4b8b8fcc23ffc3a773a08bf25819
KyberNetworkProxyInterface.sol	03e88d920e4718e4eafe28f5887643c87e07e079f5d94478495d6a3f3f872430
KyberReserve.sol	759503e0ace2ca27c3ed2902d9ec6a12d8351c4f3bf14ad9e44c9c5d5a30398d
KyberReserveInterface.sol	eff68a389631936387fae004bdef82fcf4bbdb62ee2bcb4c9570b14c15899d62
PermissionGroups.sol	b9049798efd7f635369e27552ef6cf33265c23b440c7bb0336d64978df8f87fa
SanityRates.sol	21eb92652d1f1fcb335de0e19c087c2b1bbf2837a58c115f8fffedb891070de2
SanityRatesInterface.sol	2ab77302b357d7f6922c465744ab3354ef1b713b43176bd705b7483fed8cda36
SimpleNetworkInterface.sol	9d0950410c2028319a8027fb51e546430b9bdac1c3e7c297b80d7d94b560ba31
Utils.sol	85a24703baf20337be2642d2fcdd2596e020c5038431e6cae4195629e0f32700
Utils2.sol	10ae99254872f55e3f2d91eb003d382efae9874b7d45535ae74ae85b46eadafe
VOLUMEImbalanceRecorder.sol	59f9e0b739ad93644844963d99d96868ef9f3bb47dd88fa0ad66c3418c423103
WhiteList.sol	a14edadddd491616dc68c9300db50de7c9b4c8daeedb1f007e25ccc7a1db80ed
WhiteListInterface.sol	0e4e871e3a01d9ff771b83797e6183501cff2b54cea2c8513ae4e8a950ffec3c
Withdrawable.sol	fd9ddf8bee9eec20abc55767138e2815f2b9ac2b39807a54642f069967670f10

Depth of Audit

The scope of the security audit conducted by CHAINSECURITY was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

²Git commit: 9a7f78c77d8946ed027fa8669a180e8e7c203cc8

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology³).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

We categorize the findings into 4 distinct categories, depending on their severities:

-  Low: can be considered as less important
-  Medium: should be fixed
-  High: we strongly suggest to fix it before release
-  Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

LIKELIHOOD	IMPACT		
	High	Medium	Low
High			
Medium			
Low			

During the audit concerns might arise or tools might flag certain security issues. If our careful inspection reveals no security impact, we label it as  **No Issue**. If during the course of the audit process, an issue has been addressed technically, we label it as  **Fixed**, while if it has been addressed otherwise by improving documentation or further specification, we label it as  **Addressed**. Finally, if an issue is meant to be fixed in the future without immediate changes to the code, we label it as  **Acknowledged**.

Findings that are labelled as either  **Fixed** or  **Addressed** are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

³https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

Scope

In cooperation with KYBER.NETWORK, CHAINSECURITY derived a specification for the system. This specification determines the audited system properties and furthermore defines what is inside and outside the scope of the audit.

Included in the scope

Main Properties

These properties are discussed in more detail in the Trust Model.

- User funds, including ether and tokens but excluding gas payments, are not at risk even if KYBER.NETWORK admins and reserves are malicious
- Reserve funds are not at risk even if Kyber.Network admins are malicious (Only KNCs in the fee wallet are at risk)

Properties during a sale

- The network moves user funds to the reserve offering the best rate
- The reserve is correctly charged (in burnt tokens)
- Users cannot exceed their cap
- The chosen reserve has enough tokens/ETH for the trade
- The chosen reserve's conversion rate is above the user specified one
- The reserve has to deliver the correct amount of tokens to the Network, which forwards it to the user⁴
- Token to token exchanges use ETH as an intermediate

Reserves

- The recorded amount of to-be-burnt KNCs is correctly computed
- The burning is not enforced and requires reserve cooperation
- The amount of to be burnt tokens is correctly updated after each trade
- Sufficient deposit at all times of KNC is currently not enforced
- Reserves need to be able to update exchange rates
- Integrity of exchange rates needs to be preserved
- Reserves are expected not to selectively block transfers, in case they do they will be delisted
- A big imbalance will block future requests
- If a price update transaction took more than 5 blocks to be confirmed, the imbalance is imprecise
- Imbalance is the imbalance since the last confirmed price update transaction
- $\text{VolumeImbalance} = \text{NumTokensSoldByReserve}(\text{Token X}) - \text{NumTokensBoughtByReserve}(\text{Token X})$
- Reserve should be able to fetch funds from any ETH address using transferFrom
- Should support a step function that offers different rates according to different order quantities
- There are sanity rates, which protect the reserves when market deviates > 12.8% in either direction.
- A block number is attached to every price update

⁴Ignoring minor rounding differences.

Users

- Users are subject to a user cap (set according to multiple criteria) e.g, users who did KYC will get higher cap
- Unfulfillable trade requests (due to high limit) will fail
- Users are subject to a gas price limit

Proxy Contract

- The proxy contract allows upgradability with respect to the network
- However, the proxy contract still provides some trust
- Upgradability needs to be tightly guarded
- The proxy ensures correct conversion rate (minimum given by user or better)

Network

- Controlled by a Multi-Signature Contract
- Network admin can list and unlist tokens and reserves
- `getExpectedRate` gives correct estimates
- Network admin can set fees
- But fee payment is not enforced
- The network contract does not hold major funds (of tokens or Ether) during regular usage
- Minor funds can accumulate due to rounding errors in arithmetic computations
- Exception is the digix token, cf out of scope section

Miscellaneous

- Arithmetic calculations cannot result in under/overflows
- Arithmetic calculations have appropriate precision
- Constraints on arrays exist and are correctly enforced
- There cannot be locked tokens or ETH ending up in contracts where they cannot be handled

Important Notes

- KYBER.NETWORK intentionally did not change compiler version from v1
- The KNC-ETH rate is manually inserted (updated as needed due to price volatility)

Out of Scope

- Checks for the gas efficiency of listed tokens (currently done off-chain)
- KYC process of Reserve Managers (done off-chain)
- All contracts in the folders `mockContracts` and `wrappers` and the `Abi` folder
- Correct use of the different contracts (e.g. no modifications) (checked off-chain)
- Website integration including all dApps
- Multi-Signature contract controlling the `Kyber.Network`
- The special case of the digix token, where transfer fees are paid to the digix dao

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover 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 lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed auditing in order to discover as many vulnerabilities as possible.

Details of the Findings

In this section we detail our results, including both positive and negative findings.

Security Issues

In the following, we discuss our investigation into security issues. Therefore, we highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

1.1 Kyber Reserve ✓ No Issue

CHAINSECURITY checked the following points related to the KyberReserve contract. The subpoints listed below are comments on the fulfillment of the specification.

- Burning is eventually enforced.
 - A public function `burnReserveFees` in the `FeeBurner` contract exists.
 - At this point a reserve can avoid paying the fee, which should result in it being delisted manually by the network administrator.
- The amount of tokens to be burnt is correctly updated after each trade.
 - The `handleFees` and `burnReserveFees` functions are the only functions modifying `reserveFeeToBurn`. `handleFees` gets called at the end of the trading function in the `KyberNetwork`.
 - No taxation happens on ETH-to-ETH trades.
- Sufficient deposit at all times of KNC is currently not enforced.
 - Currently there is no such mechanism available.
 - The balance of a reserve with regards to a specific token can be queried with the public function `getBalance`.
- Reserves need to be able to update exchange rates.
 - This can be done by calling `setBaseRate` and `setQtyStepFunction`.
- Integrity of exchange rates needs to be preserved.
 - Fulfilled under the assumption that integrity means that only allowed actors can change rates.
 - Both `setBaseRate` and `setCompactData` can only be updated by `onlyOperator`.
- A malicious administrator can only cause the loss of the KNC of the reserve.
 - Not just KNC but any token accidentally trapped in the reserve contract through the payable function, by setting himself as an operator and approving withdrawal.
- A big imbalance will block future requests.
 - Enforced by returning an exchange rate of 0 in such a case in the `ConversionRates` contract, `getRate` function.
- Reserve should be able to fetch funds from any ETH address using `transferFrom`.
- Should support a step function that offers different rates according to different order quantities.
 - Implemented in the `ConversionRates` contract with a struct `StepFunction`.
 - Each struct `TokenData` has the following `StepFunction` fields: `buyRateQtyStepFunction` and `sellRateQtyStepFunction`.
 - The above fields are set by `onlyOperator`.
- There are sanity rates, which protect the reserves.

1.2 Volume Imbalance ✓ No Issue

CHAINSECURITY verified the following specification related to the volume imbalance:

- A block number is attached to every price update.
- The imbalance is the imbalance since the last confirmed price update transaction.
- If the price update transaction took more than 5 blocks to be confirmed, the imbalance is imprecise.
- The volume imbalance of a specific token is calculated as the number of such tokens bought by the reserve subtracted from the number of such tokens sold by the reserve.

Additionally, the diagram below illustrates how the exchange rate interacts with the volume imbalance in a simplified fashion, highlighting key parameters and functions.

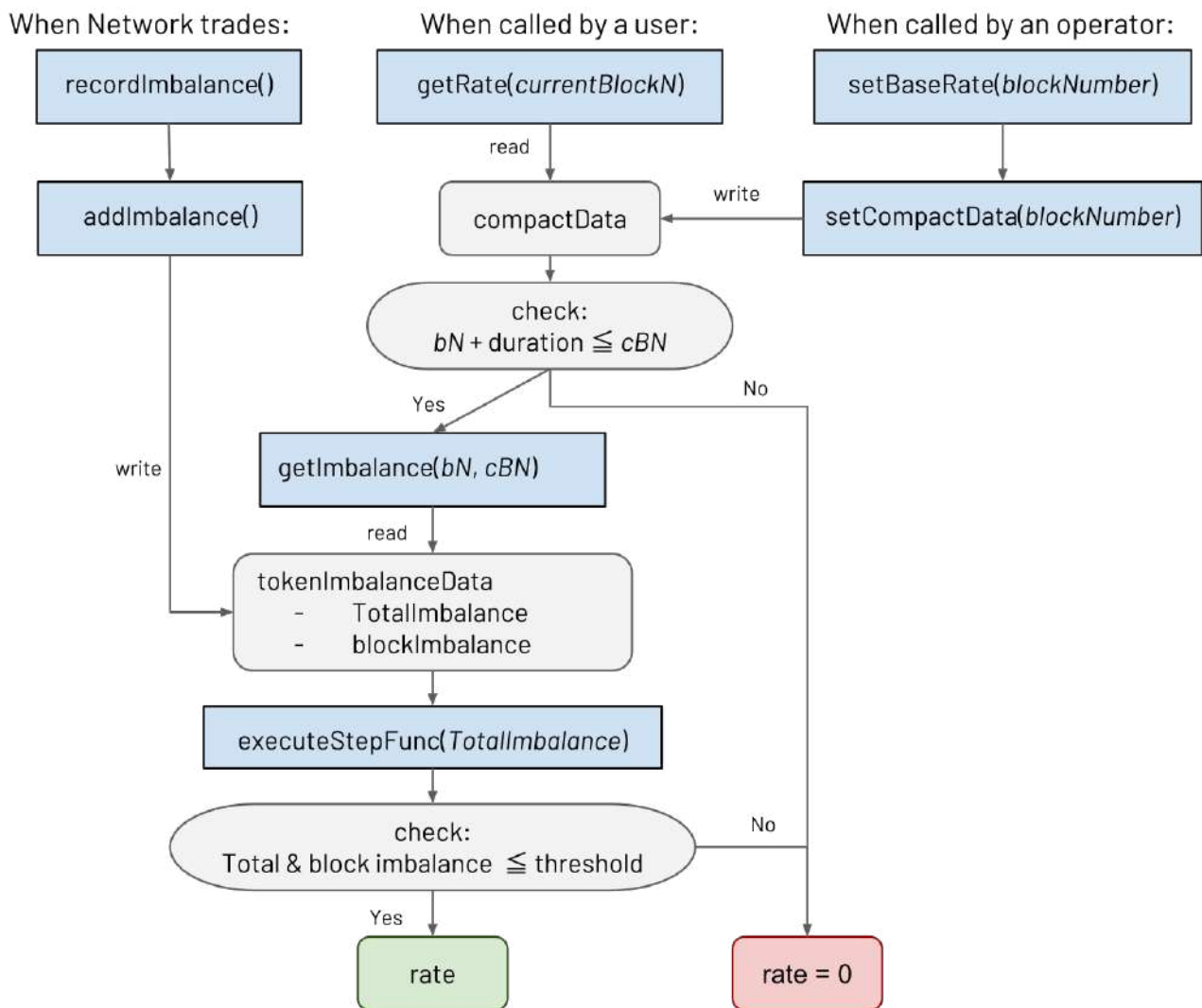


Figure 1: Rate and imbalance interaction for an arbitrary token.

1.3 User requirements ✓ No Issue

CHAINSECURITY did not find issues in the specific points:

- User cap
- User can only lose gas or time due to malicious behavior
- User only needs to review the `KyberNetworkProxy.sol` file and the token contract
- Unfulfillable trade requests (due to high limits) fail User has gas price limits

We now explain these points in detail.

The admin (i.e. smart contract owner) of the Proxy contract can connect it to any contract that matches the `KyberNetworkInterface`. This is the main purpose of a proxy contract, to be upgradeable.

But this also represents a threat in terms of trust, since the admin may change the underlying contract at any time to an unaudited and potentially buggy or malicious version.

The `KyberNetworkContract` variable is public, and its value can be easily checked if the code has been published/verified (as it undoubtedly will be).

As we will see below, the proxy is built in such a way to make the review of the behind the scenes contracts irrelevant to the user. Indeed, once ETH or tokens are entrusted to the proxy for a transaction, they will either be traded for the requested tokens (including ETH), respecting the minimum expected rate, or refunded through a revert (which rolls back to the state prior to the trade). It is therefore sufficient for the user to review and establish trust in the proxy contract, without worrying about the implementation details behind it.

The proxy contract inherits from 3 other smart contracts: one interface (which helps other contracts interact with it), `Withdrawable.sol` and `Utils2.sol`, both adding some utility features.

`Utils2` inherits from the `Utils` contract. Together, they define a set of functions to handle retrieval and calculation of decimals (which may differ from token to token), user balance of ETH and tokens, conversion rates and amounts of destination and source tokens. All functions in `Utils` and `Utils2` are consultative and do not change the state of the blockchain, except for one, which keeps track of the decimals of each token.

`Withdrawable` makes sure no ETH nor tokens can get stuck in the contract, and allows the admin to move them wherever he chooses. `Withdrawal` functions can only be called by the admin and only externally, so they can never interfere during a transaction (it would require them to be called internally, which is forbidden).

The `trade` and `tradeWithHint` functions are payable (i.e. they accept ETH), but the received ETH is immediately sent to `KyberNetwork` in the same transaction, and either traded for tokens or refunded through a revert (in case the transaction fails for whatever reason). The same is true for token to ETH trades. This means the `Withdrawable` behavior can never be used to steal ETH (nor tokens) from trades, since those valuables are never stored in the contract itself in between transactions.

The proxy contract forwards all trade transactions to the `KyberNetwork` contract, along with a reference to the trader's address, and verifies that the result of the transaction is in line with expectations. All other functions are direct proxies to their equivalents at the `KyberNetwork.sol` level, except for `calculateTradeOutcome`, which is used as a verification tool for trades.

The verification is done as follows: the trade is executed and the network reports a destination amount (tokens that should have been received as result of trade). The `calculateTradeOutcome` function is then used to observe the actual amount received by the user, in a way that is totally independent from the `KyberNetwork` contract implementation (current or future). Two conditions might make the transaction fail, and thus revert all operations to the state prior to the trade:

- If the reported and observed amounts do not exactly match
- If the amount is not equal or greater than the minimum expected amount (derived from the minimum rate initially mentioned in the transaction)

This means that whatever happened behind the scenes (e.g. in `KyberNetwork.sol` or other related contracts), the destination amount of tokens (or ETH) will respect expectations, or cancel the trade (with gas fees expenditure as the only consequence for the user).

One last piece of the system might present a risk to the user, and that is the behavior of the tokens involved in the trade. Neither `KYBER.NETWORK` nor the reserve managers can take responsibility for the behavior of those tokens, so it is up to the user to establish if the token smart contract can be trusted. This is outside the scope of this audit, and the user is encouraged to verify the trustworthiness of tokens he wishes to trade, by reviewing their smart contract code or the related audits.

1.4 Network requirements (partial) ✓ No Issue

Some points of the specification we checked had issues which we explained in other parts of this report. The network is under control of an admin, and users should make sure that this admin is actually a multisig that cannot be compromised by a rogue admin. `CHAINSECURITY` did not find issues in the following points:

- The network has the ability to list and unlist tokens and exchanges
- `getExpectedRate` gives correct estimates
- During regular usage, the token and ETH balances are null after a transaction.

1.5 Overflow in `reserveFeeToBurn` possible L

The FeeBurner contract uses the function `handleFees` to keep track of reserves' fee debts. However, through the unsafe addition `reserveFeeToBurn[reserve] += feeToBurn`, a reserve can purposefully overflow the counter to avoid paying fees, for example by trading with itself.

However, since a reserve would need to pay for the gas costs of a lot⁵ of transactions to achieve the overflow, this is extremely unlikely.

Likelihood: Low

Impact: Low

1.6 Overflow in `getImbalance` possible L ✓ Acknowledged

In the `getImbalance` function, the multiplication `totalImbalance *= resolution` is performed unchecked. Since the `resolution` field can be arbitrarily big and `totalImbalance` can reach $2^{63}-1$, overflows are possible. A check on writing the `resolution` variable could be introduced to avoid this.

Since the function is only used to retrieve information and in a benevolent case `resolution` should not be this big, this is unlikely. But because this function is used in the `getRate` function, a consequence could be returning a wrong rate to users.

The same applies to `currentBlockImbalance *= resolution`.

Likelihood: Low

Impact: Low

Acknowledged: KYBER.NETWORK is aware of this issue.

1.7 Influence of `maxDestAmount` on chosen conversion rate M ✓ Acknowledged

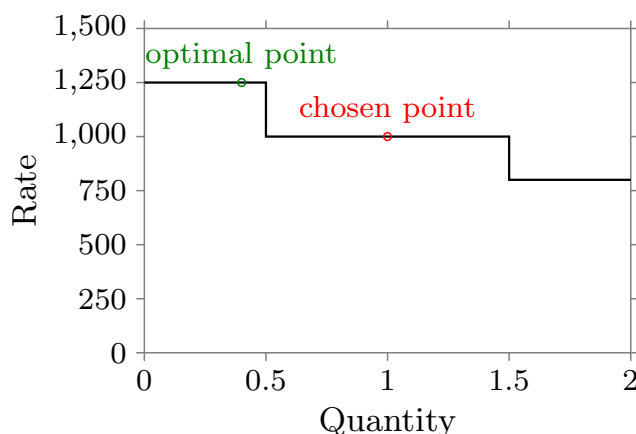
When trading, a user can specify the `maxDestAmount`, which signals the maximum amount of output tokens it aims to receive. In the current design, this amount is only considered after the best rate has been chosen, therefore the chosen rate can be "incorrect" in different ways:

1. *Too low rate due to wrong reserve:* The user might receive a smaller rate than possible because the wrong reserve was chosen due to the non-consideration of `maxDestAmount`. Example:

- Trade request: 1 ETH $\rightarrow$ KNC, `maxDestAmount` = 500 KNCs
- Reserve A: rate = 500, 1000 KNCs
- Reserve B: rate = 1000, 500 KNCs
- Reserve B is chosen, because reserve A has insufficient funds for 1 ETH.

In the example above, reserve B could have offered a better rate.

2. *Too low rate due to cached rate:* The user might receive a smaller rate than possible due to the influence of `maxDestAmount`. Consider the conversion rate of an exchange as shown below:

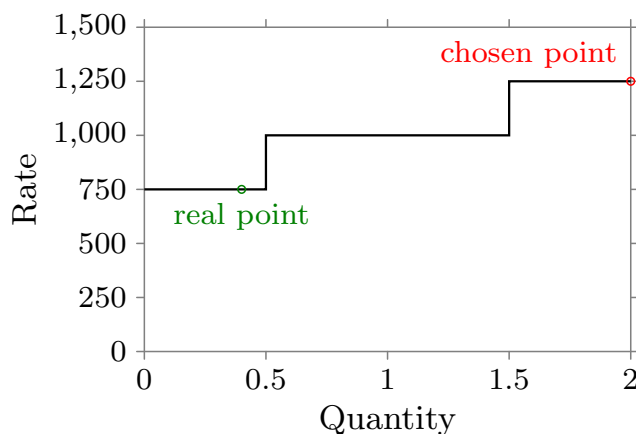


⁵Assuming a `walletFee` of 0, maximum `reserveFeesInBps`, `kncPerETHRate` and `tradeWeiAmount` this would be on the order of 10^{27} transactions.

In case of a trade request for 1 ETH $\rightarrow$ KNC, $\text{maxDestAmount} = 500$ KNCs, the chosen rate will be 1000, as the red dot is the chosen point on the curve. Afterwards, only 500 KNCs will be emitted resulting in a price of 0.5 ETH.

In the optimal case, the green point would have been chosen, as it suffices to exchange 500 KNCs. In that case the rate would have been 1250 and hence the price would have been 0.4 ETH.

3. *Too high rate due to cached rate*: In case an exchange exists a reserve which has a (partially) increasing conversion rate in relation to the exchanged source quantity, then the user can reach a better rate using maxDestAmount . Consider the conversion rate of an exchange as shown below:



In case of a trade request for 2 ETH $\rightarrow$ KNC, $\text{maxDestAmount} = 500$ KNCs, the chosen rate will be 1250, as the red dot is the chosen point on the curve. Afterwards, only 500 KNCs will be emitted resulting in a price of 0.4 ETH.

In the optimal case, the green point would have been chosen, as it suffices to exchange 500 KNCs. In that case, the fair rate would have been 750 and hence the price would have been 0.66 ETH.

Hence, the user is able to enforce a higher rate than expected for reserves with a (partially) increasing conversion rate by making a trade request with a large source amount.

Likelihood: Medium

Impact: Medium

Acknowledged: KYBER.NETWORK is aware of this issue. KYBER.NETWORK will advise the reserves to avoid the use of increasing exchange rates to omit the third issue. The avoidance of the first two issues is left to the user and its application. While it cannot be entirely resolved due to the inherent non-atomicity of multiple blockchain interactions, KYBER.NETWORK will advise applications on how to reduce the effect of this issue. Finally, improvements with respect to this issue will be made in next version of the project.

1.8 Failing transaction due to rounding issue M ✓ Acknowledged

During the audit, the following code was modified:

```

125     require(reportedDestAmount == tradeOutcome.userDeltaDestAmount);
126     require(tradeOutcome.actualRate >= minConversionRate);
127
128     ExecuteTrade(msg.sender, src, dest, tradeOutcome.userDeltaSrcAmount, tradeOutcome.userDeltaDestAmount);
129

```

KyberNetworkProxy.sol

This new code can lead to a failure of a valid transaction. To explain, we provide the following example:

- One exchange offers an ETH-KNC rate of 1.1
- A user tries to exchange 300 wei into KNC with a minimum conversion rate of 1.1 and a maximum destination amount of 218 KNC wei.
- Due to the maximum destination amount, only 199 wei will be exchanged.
- Hence, the `actualRate` computed for the line above will be $\frac{218}{199}$ and therefore smaller than the minimum conversion rate of 1.1.

- The proxy reverts the transaction.

As the example shows, a satisfiable transaction can fail, due to rounding issues. The likelihood of occurrence depends on the frequency with which the maximum destination amount is set and the conversion rates of the reserves.

Likelihood: Low

Impact: Medium

Acknowledged: KYBER.NETWORK considers that this behaviour is unlikely to occur, based on the previous usage patterns. Additionally, it will advise integrating application to use slightly lower minimum conversion rates in order to avoid the issue.

1.9 Locked tokens or ETH

Thanks to the nice design of the `Withdrawable` contract, any token sent should be retrievable by KYBER.NETWORK, and sent back to the rightful owner.

Trust Issues

The issues described in this section are not security issues but describe functionality which is not fixed inside the smart contract and hence requires additional trust into KYBER.NETWORK, including in KYBER.NETWORK ability to deal with such powers appropriately.

2.1 Wrong require in `setMinSlippageFactor` can lead to Underflow

In `ExpectedRate.sol`, the function `setMinSlippageFactor` sets the variable `minSlippageFactorInBps` to the input `bps`.

```
function setMinSlippageFactor(uint bps) public onlyOperator {
    require(minSlippageFactorInBps <= 100 * 100);

    MinSlippageFactorSet(bps, minSlippageFactorInBps, msg.sender);
    minSlippageFactorInBps = bps;
}
```

ExpectedRate.sol

`minSlippageFactorInBps` must be less than or equal to 10'000, because otherwise the following could cause an underflow:

```
minSlippage = ((10000 - minSlippageFactorInBps) * expectedRate) / 10000;
```

ExpectedRate.sol

The problem is the `require` in `setMinSlippageFactor`. It checks whether the old value is less than 10'000, not whether the new value is less than 10'000. This means that not only can it be set to a large value causing an underflow, but also because of this `require` statement, once `minSlippageFactorInBps` has been set to this wrong value, it cannot be changed again, effectively breaking the code.

For this issue to arise, the operator would have to make a mistake when using this function, which makes the whole issue unlikely.

Fixed: The correct `require` is now in place.

2.2 Additional Proxy Checks possible

As described above, the proxy is supposed to protect the user. For that purpose, the proxy could perform the following additional checks.

1. Enforce maximum destination amount: The proxy could enforce that no more than the desired maximum was generated.
2. Check for empty transactions: The proxy currently contains the following checks:

```
188     require(userDestBalanceAfter >= destBalanceBefore);
189     require(srcBalanceBefore >= userSrcBalanceAfter);
```

KyberNetworkProxy.sol

In case, if these checks used `>` instead of `>=`, they could filter out empty transactions and hence prevent the subsequent exception through a division by zero, which consumes all of the user's gas.

Fixed: KYBER.NETWORK added additional checks inside the proxy to further protect the user. These checks also include a check for the maximal destination amount:

```
194         require(tradeOutcome.userDeltaDestAmount <= maxDestAmount);
```

KyberNetworkProxy.sol

Design Issues

The points listed here are general recommendations about the design and style of KYBER.NETWORK's project. They highlight possible ways for KYBER.NETWORK to further improve the code.

3.1 Capitalization of constants

Constants should be capitalized. An example violating this convention is the constant `public kgtHolderCategory` in the Whitelist contract.

3.2 Specification mismatch

KYBER.NETWORK listed the following point as a part of the specification:

- When the market deviates more than 12.8% in either direction the base price should be updated.

However this does not hold, since currently price updates only happen manually.

Addressed: The specification has been revised to indicate that the base rate update is manual.

3.3 `validateTradeInput` should fail by default

`validateTradeInput` checks several conditions, and returns true by default. The other pattern is preferable, i.e. it is better to fail by default, in case a case has been forgotten in the implementation.

On top of that, given the current use of this function, the function could revert if a bad input is encountered, rather than returning the corresponding boolean. It would reduce the boilerplate and make the function clearer.

Fixed: The checks are now implemented using a series of `require`.

3.4 Linter warnings disabled with no explanation

Solhint code complexity warnings are disabled in some functions, and indeed these functions are complex and would benefit from being split into several internal functions. Warnings should be heeded rather than ignored in these cases, or at least, further justification should be provided as to why they have been ignored.

This also applies to other patterns that are deactivated in the code.

Addressed: KYBER.NETWORK added comments, where applicable. Some functions could still be split into several for better readability.

3.5 `transferAdminQuickly` bypasses sanity checks

KYBER.NETWORK has implemented a function to transfer the rights to a new administrator without any check.

```
/**
 * @dev Allows the current admin to set the admin in one tx. Useful initial deployment.
 * @param newAdmin The address to transfer ownership to.
 */
function transferAdminQuickly(address newAdmin) public onlyAdmin {
    require(newAdmin != address(0));
    TransferAdminPending(newAdmin);
    AdminClaimed(newAdmin, admin);
    admin = newAdmin;
}
```

Since there is no limit on this function, this can be reused even after initial deployment, which is bound to happen.

To prevent such a misuse, this function should be removed. Setting the administrator in the initial deployment can be done through the constructor instead. Even if set manually, this means that only two transactions have to be done instead of one.

Acknowledged: KYBER.NETWORK acknowledges the issue, but only plans to use the function during deployment to save several costly multisig transactions. Thereby, KYBER.NETWORK plans can reduce the potential risk.

3.6 Code consistency

The file `PermissionGroups.sol` contains both

```
alertersGroup.length—;
```

`PermissionGroups.sol`

and

```
operatorsGroup.length -= 1;
```

`PermissionGroups.sol`

Only one of these forms should occur in the code, to make it more readable.

Another consistency issue arises with if-branches that contain only one statement. KYBER.NETWORK's code contains the following:

```
if (destAddress != (address(this)))
    destAddress.transfer(amount);
```

`KyberNetwork.sol`

as well as:

```
if (src == ETH_TOKEN_ADDRESS) {
    callValue = amount;
}
```

`KyberNetwork.sol`

Only one style should be used, either use curly braces everywhere (recommended), or do not use curly braces when there is only one statement.

3.7 Inconsistent use of `feeAmount` in `sendFeeToWallet`

The `sendFeeToWallet` function is designed as follows:

```
125 function sendFeeToWallet(address wallet, address reserve) public {
126     uint feeAmount = reserveFeeToWallet[reserve][wallet];
127     require(feeAmount > 1);
128     reserveFeeToWallet[reserve][wallet] = 1; // leave 1 twai to avoid spikes in gas fee
129     require(knc.transferFrom(reserveKNCWallet[reserve], wallet, feeAmount - 1));
130
131     feePayedPerReserve[reserve] += (feeAmount - 1);
132     SendWalletFees(wallet, reserve, msg.sender);
133 }
```

`FeeBurner.sol`

One twai is left inside the fee balance to avoid higher gas prices for the next trader interacting with this reserve. Therefore, `feeAmount - 1` is being transferred. Hence the `require` should also check `feeAmount - 1` which is why the assignment in line 126 could directly be

```
126     uint feeAmount = reserveFeeToWallet[reserve][wallet] - 1;
```

to ensure consistency.

Additionally, to avoid the same issue and increase code readability, the `burnReserveFees` function can set `uint burnAmount` to `reserveFeeToBurn[reserve] - 1` directly, saving writing `burnReserveFees - 1` several times later on.

3.8 Lack of indexed fields in some events

Some events do not have any indexed field, although it seems natural that they should have one. Examples are `TokenWithdraw` and `EtherWithdraw` in `Withdrawable.sol`, but there are others. KYBER.NETWORK should review all events and make sure that fields are indexed when appropriate.

Acknowledged: KYBER.NETWORK is aware of this issue.

3.9 Off-by-one error with MAX_QTY

Trading exactly MAX_QTY seems to be prohibited by the validation function:

```
function validateTradeInput(ERC20 src, uint srcAmount, address destAddress) internal view returns(bool) {  
    if ((srcAmount >= MAX_QTY) || (srcAmount == 0) || (destAddress == 0))  
        return false;  
}
```

KyberNetwork.sol

However other functions treat this case as a valid one. The tests on `getExpectedRate` also imply that only `MAX_QTY + 1` is prohibited.

Fixed: Trading exactly MAX_QTY is now possible.

3.10 Assigning to function parameters

Assigning to function parameters should be avoided. Two occurrences of this can be found in KYBER.NETWORK, specifically in the `ConversionRates` contract:

- In the `recordImbalance` function, the argument `rateUpdateBlock` is written to.
- In the `getRate` function the unsigned integer `qty` is overwritten.

Acknowledged: KYBER.NETWORK is aware of this issue.

3.11 Gas savings possible

In a number of places, a `uint` is checked for positivity, e.g. `uint x > 0`. However, in the case of `uint`, the check `x > 0` is equivalent to `x != 0`, as `uint` cannot hold negative values. The latter check is significantly more gas-efficient in both deployment and execution costs and should hence be preferred for gas efficiency. This would however require more care if KYBER.NETWORK ends up changing the type of these variables to `int` rather than the current `uint`, given that this would make negative values possible.

Fixed: KYBER.NETWORK is aware of this possible optimization.

3.12 Front-running possible

As in any marketplace or exchange, users can influence each other's transactions by design. In particular, in KYBER.NETWORK users can try to front-run each other in order to receive arbitrage amounts.

In case of a sparse order book, a reserve could front-run major purchase transactions, buy the funds and directly offer them again at a higher price which results in the chosen minimum conversion rate of the front-run transaction. Depending on the chosen parameters and the structure of the market, different gains would be possible in such a behavior.

Acknowledged: KYBER.NETWORK has acknowledged the existence of these issues. As previously, this issue is inherent to marketplaces and exchanges and was already partially addressed with the minimum conversion rate.

Recommendations / Suggestions

- ☒ In `ExpectedRate.sol`, the variable name `minSlippage` is confusing. We would recommend replacing it with something like `expectedWorstCaseSlippageRate`.
- ☒ In `KyberNetworkProxy.sol`, a comment says “make sure no overflow”, however, the associated check prevents an underflow.
- ☒ In the contracts, sometimes ETH is referred to as such (e.g. `ethAmount`) and sometimes (more correctly) as amount in wei (e.g. `tradeWeiAmount`). Consistency could aid the general readability.
- ☐ Several unused imports exist in the codebase. Concretely, the `FeeBurner` contract imports the `ERC20Interface`, but does not use it. These imports are unnecessary and misleading.
- ☐ The visibility of some functions should be further restrained, so that their use is clearer when reading the code. In particular, the functions `setReasonableDiff` and `setSanityRates` in the `SanityRates` contract should be specified as external.
- ☐ Additionally, our testing showed that calling either of these two functions with more than 363 tokens would result in exceeding the block gas limit⁶ and failure. We recommend to document this for future reserve managers.
- ☐ `getUserCapInTokenWei` in `KyberNetwork.sol` is not implemented, but returns 0. To prevent any misuse, a `assert(false)` is more appropriate.
- ☐ Testing revealed that the `ConversionRates` contract function `setBaseRate` exceeds a block gas limit of 8 million when called with approximately 180 tokens. This information can be added to the documentation.
- ☐ `ETH_TOKEN_ADDRESS` could be renamed to `ETH_TOKEN` given that it is used as an ERC20 token.
- ☐ The `VolumeImbalance` contract defines two structs `TokenControlInfo` and `TokenImbalanceData`. The fields of these structs could be restricted to smaller `uint` sizes, to avoid overflows as mentioned in the issues and save on storage and gas.
- ☒ The variable `buy` is defined twice inside the `KyberReserve` contract, once as a `bool` indicating the trade direction, in the `getConversionRate` function, and once as an `int` in the `doTrade` function, denoting the signed trade amount. For consistency and better readability, we recommend to rename one of the variables, e.g. to `tradeVolume` for the integer version.
- ☐ Unnecessary type conversions are introduced in the `VolumeImbalance` contract. In the `addImbalance` function, the argument `uint currentBlock` is passed and again cast to `uint` a few lines later. The same applies to the `int buyAmount` argument.
- ☐ In the `KyberReserve` contract, `doTrade` function, a boolean parameter `validate` allows to skip some require statements, presumably to save gas on duplicate checks. These savings are rather small and the code could be simplified by just verifying twice, which is less error-prone.
- ☐ Basic rate steps (`bps` variables) are by design limited to $100 * 100$. `KYBER.NETWORK` could use a more systematic check on these variables, in the current state there is little consistency between the various checks.
- ☒ The `KyberNetworkInterface` could be an interface instead of a contract.
- ☒ A user can convert from currency `T` to `T`, even when `T` is ETH. This seems to offer little benefit, and `KYBER.NETWORK` has to handle this case separately, rather than just checking `assert(src != dst)`.

⁶ Assuming an 8 million limit.

Users might be tricked in case extra events are added

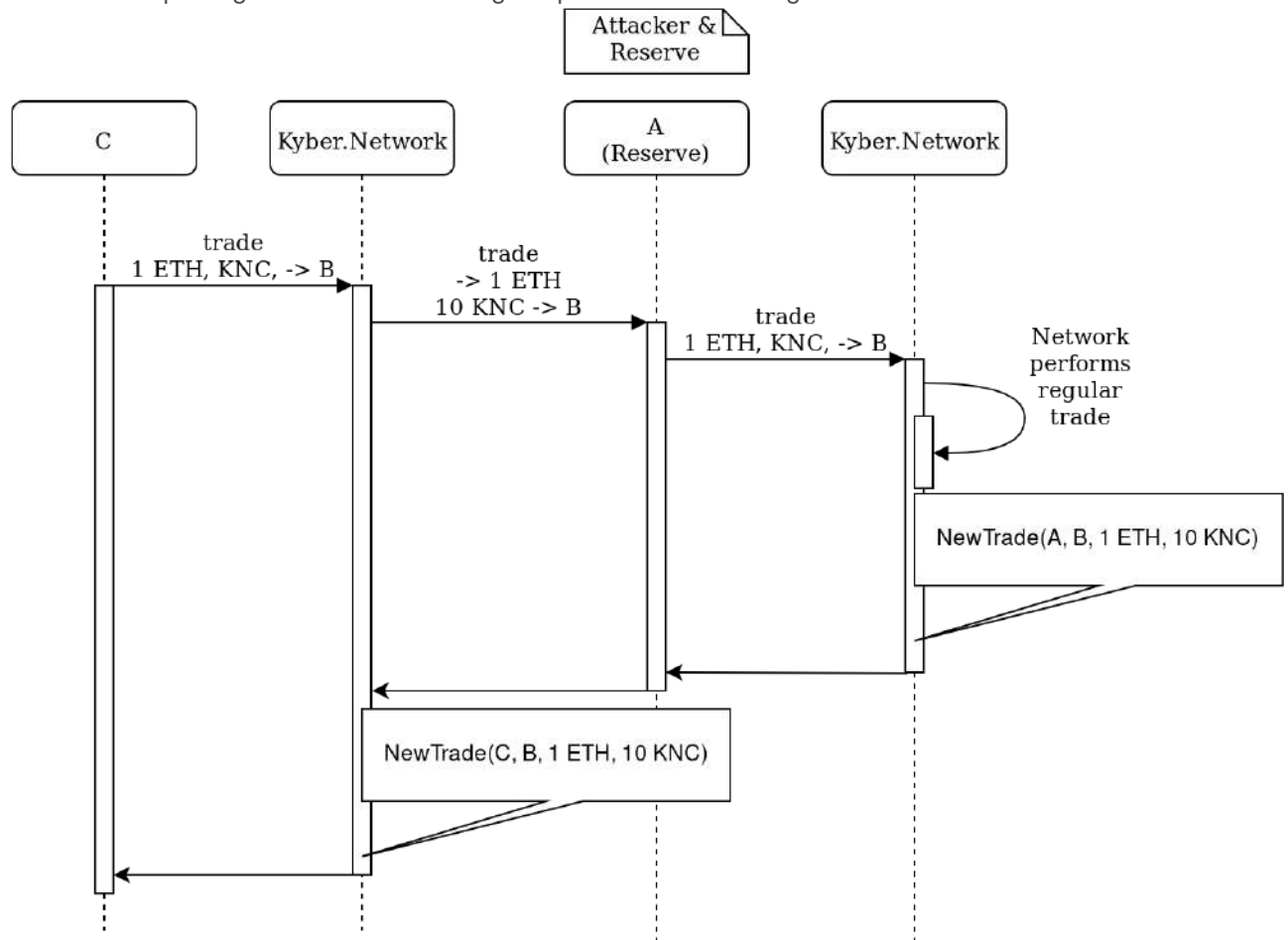
Currently, KYBER.NETWORK emits no events which contain the destination address of the trade. If such events would be added in the future, recipients could be tricked in the following way.

In case a KYBER.NETWORK user B is awaiting the payment for another KYBER.NETWORK user A and therefore monitors the `NewTrade` event which signals completed trades, then B can be tricked.

The attack scenario is as follows:

- B awaits a payment of 10 KNCs from A
- A is a reserve for KNCs
- An innocent user C happens to make a KYBER.NETWORK transaction, aiming to transfer 10 KNCs to A

A can use its privileged role as an exchange to perform the following attack:



- A receives the 1 ETH paid by C
- A uses the 1 ETH to issue a new trade inside the Kyber.Network
- The trade is performed regularly and the event `NewTrade(A,B,1 ETH, 10 KNC)` is emitted

⇒ B thinks that it received a payment from A, but A has used no funds.

- The original trade is completed and the event `NewTrade(C,B,1 ETH, 10 KNC)` is emitted

⇒ B thinks that it received a payment from C

This might happen to merchants which expect to receive multiple payments. It could also be used by clients to “double-spend” against such a merchant. If, in the example above, A and C are controlled by the same entity, they can make two payments at once.

Disclaimer

UPON REQUEST BY KYBER.NETWORK, CHAINSECURITY LTD. AGREES MAKING THIS AUDIT REPORT PUBLIC. THE CONTENT OF THIS AUDIT REPORT IS PROVIDED "AS IS", WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND CHAINSECURITY LTD. DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH CHAINSECURITY LTD..