

Security Audit

of REPUBLIC PROTOCOL's Smart Contracts

October 22, 2018

Produced for








REPUBLIC PROTOCOL




















by



CHAINSECURITY

Table Of Content

Foreword	1
Executive Summary	1
Scope	2
I. Included in the scope	2
II. Out of Scope	2
Audit Overview	3
I. Scope of the Audit	3
II. Depth of Audit	3
III. Terminology	3
Limitations	5
System Overview	6
I. System Overview	6
Best Practices in REPUBLIC PROTOCOL's project	7
I. Hard Requirements	7
II. Soft Requirements	7
Security Issues	8
I. Darknodes and darknode registry	8
1.1 Basic specification	8
1.2 Challengers might not receive rewards  	8
II. Dark pools	9
2.1 Basic specification	9
III. Order book	9
3.1 Basic specification 	9
3.2 Unfixed DoS in orderbook  	10

3.3	Silent failures			10
3.4	orderID might leak information			11
3.5	orderOpeningFee is lost in OrderBook			11
IV.	Traders and brokers			12
4.1	Basic specification			12
V.	Miscellaneous			12
5.1	Basic specification			12
5.2	Unchecked arithmetic operations			12
VI.	RenEx Settlement and fee payment			13
6.1	Basic specification			13
6.2	Funds backing orders can be withdrawn			14
6.3	Double-spending with RenEx atomic swaps			15
6.4	Malicious trader can defer call to redeem			15
6.5	Requirements for Used Tokens			16
	Trust Issues			17
I.	Bond of darknode can be abused			17
II.	All atomic swaps can be slashed			17
III.	submissionGasPriceLimit can selectively freeze functionality			17
IV.	Owner of RenExSettlement can selectively manipulate			17
	Design Issues			19
I.	Inefficient transfers			19
II.	Missing requires			19

III.	Simple storage savings possible			19
IV.	Suboptimal <code>struct</code> swap			20
V.	Redundant casts			20
VI.	<code>deposit</code> breaks Checks-Effects-Interactions pattern			20
VII.	Outdated compiler version			21
VIII.	Atomic swaps generate free options			21
IX.	Better definition of “sufficient time” for atomic swap			21
X.	<code>RenExAtomicInfo</code> should emit more events			22
XI.	Possible gas optimizations in <code>RenExAtomicSwapper</code>			22
XII.	No Input Validation for <code>submissionGasPriceLimit</code>			22
XIII.	No Input Validation for Updates			23
XIV.	Broker signature includes limited data			23
XV.	Options creatable out of <code>RenExSettlement</code>			23
	Recommendations / Suggestions			24
	Disclaimer			25

Foreword

We first and foremost thank REPUBLIC PROTOCOL for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

The REPUBLIC PROTOCOL smart contracts have been analyzed under different aspects, with a variety of tools for automated security analysis of Ethereum smart contracts and expert manual review.

In summary, we found that REPUBLIC PROTOCOL is a project of high quality, employs good coding practices and has clean, well-documented code which is impressive considering the complexity of the project.

Nonetheless, CHAINSECURITY discovered several vulnerabilities of varying severity and proposed design improvements and suggestions. All vulnerabilities were swiftly reviewed and addressed by the REPUBLIC PROTOCOL team, leading to a more resilient and efficient system.

Scope

REPUBLIC PROTOCOL requested a precisely scoped audit, meant to assess the technical foundation of REPUBLIC PROTOCOL's project in its current state. To define this scope, CHAINSECURITY listed potential points of failure and agreed with REPUBLIC PROTOCOL upon them.

Issues that have been encountered while verifying this specification have been listed, even when they were not explicitly mentioned in the specification. However, this list should not be considered exhaustive with respect to the security of REPUBLIC PROTOCOL's smart contracts. CHAINSECURITY strove to verify the points listed here, to provide a report which contents could serve as potential guidelines in the future.

The main specification sections are listed below and a detailed description of the reviewed properties can be found in the issues section.

Included in the scope

- I. Darknodes and darknode registry
- II. Dark pools
- III. Order book
- IV. Traders
- V. Miscellaneous
- VI. RenEx Settlement and fee payment

Out of Scope

- All off-chain and non-smart contract components of the system

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 August 13, 2018¹ and a updated version on October 19, 2018²:

File	SHA-256 checksum
RenExAtomicSwapper.sol	a813eb0adb542e24861182fabb178b5a4e423fa589118aa08b83bd2e49d82dbc
RenExBalances.sol	6c46428527e55c4a84a147cef10b8ea04fcc8432224171b34fba2f40337850b6
RenExSettlement.sol	bdf8b89bbd75ab713ce201dc664beb2a07ba8633b20a6a55eede3894e26b361c
DarknodeRegistry.sol	d9bd21815708f2d39a49538dd3c505066b0afaa28063d0eedabf9eb0696abd45
DarknodeRegistryStore.sol	43298b48b6e090d56ef77d476d7b76c3a3819363c000511b34b335bfa5c12bc4
DarknodeRewardVault.sol	c776949c2df68568c9f0b5b35adbe62a458937aef5d8e852d2033c628f9333cd
DarknodeSlasher.sol	5e7f90bf37efe8b7b7ede378eacf49dfdf781d566a35114d58f19f53444dfac6
SettlementUtils.sol	410a5bfa6990af0777c6ddf3a0f53e5eda5b83b6991fafba83951639d1295bfb
LinkedList.sol	820abc8bb09fab4ffd477d8bea10e9ed64e6fb6559bd459d500b100ac4064565

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.

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










¹ <https://github.com/republicprotocol/renex-sol/tree/f091d2e15068e20b843920b3767fb56ff73d0a60>, <https://github.com/republicprotocol/republic-sol/tree/10d417012aeb14aa41c04553b048e31fbd695137>

² <https://github.com/republicprotocol/renex-sol/tree/250062d32d930715e7ad6cc6d66c128a54e1dd9e>, <https://github.com/republicprotocol/republic-sol/tree/948666de5d39a55cd844cb66d6e76d3f5269c38b>





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



-  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. After careful inspection of the potential security impact, we assign the following labels:

-  **No Issue** : no security impact
-  **Fixed** : during the course of the audit process, the issue has been addressed technically
-  **Addressed** : issue addressed otherwise by improving documentation or further specification
-  **Acknowledged** : issue is meant to be fixed in the future without immediate changes to the code

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.

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.

System Overview

Token Name & Symbol	REN TOKEN, REN
Decimals	18 decimals
Tokens issued	1'000'000'000
Token Type	ERC 20
Pausable	Yes
Burnable	Yes

Table 1: Facts about the REN token.

In the following we describe the REPUBLIC PROTOCOL system. Table 1 gives the general overview of its REN TOKEN (REN).

System Overview

REPUBLIC PROTOCOL is a decentralized open-source dark pool protocol facilitating atomic swaps between cryptocurrency pairs across the Bitcoin and Ethereum blockchains. Trades are placed on a hidden order book and are matched through an engine built on a multi-party computation protocol.

While REPUBLIC PROTOCOL's complete system is beyond the scope of this audit and requires a more elaborate introduction than a brief system overview, the following list offers an overview of the reviewed contracts and their functionalities:

- Republic Solidity contracts
 - DarknodeRegistry: responsible for de-/registration of nodes
 - DarknodeRegistryStore: stores data and funds for the DarknodeRegistry
 - DarknodeSlasher: allows to challenge order confirmations and executes penalization
 - DarknodeRewardVault: holds fees for darknodes for settling orders
 - LinkedList: a library for a circular double linked list
 - SettlementUtils: a library for calculating and verifying order match details
- RenEx Solidity contracts
 - RenExBalances: holds traders funds and is used by the settlement contract/layer
 - RenExSettlement: provides on-chain settlement and fee payment for atomic swaps
 - RenExAtomicSwap: implements the RenEx atomic swapping interface for ether values

Additionally, CHAINSECURITY highlights the following roles which have special powers in this system:

- Darknode: Can participate in the protocol by registering or deregistering itself, submitting order matches, challenge matches of others. Nodes work in pools and consolidate the network.
- Slasher: A dedicated contract allowing to penalize provably malicious nodes through financial incentives.
- Owner: The owner role refers to the owner of the DarknodeRegistry and its corresponding store. This is the highest power in the system and can control the network by deciding which node to register or not and setting the token contracts.

Best Practices in REPUBLIC PROTOCOL's project

Projects of good quality follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines.

Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit.

We now list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when REPUBLIC PROTOCOL's project fitted the criterion when the audit started.

Hard Requirements

These requirements ensure that the REPUBLIC PROTOCOL's project can be audited by CHAINSECURITY.

- ✓ The code is provided as a Git repository to allow the review of future code changes.
- ✓ Code duplication is minimal, or justified and documented.
- ✓ Libraries are properly referred to as package dependencies, including the specific version(s) that are compatible with REPUBLIC PROTOCOL's project. No library file is mixed with REPUBLIC PROTOCOL's own files.
- ✓ The code compiles with the latest Solidity compiler version. If REPUBLIC PROTOCOL uses an older version, the reasons are documented.
- ✓ There are no compiler warnings, or warnings are documented.

Soft Requirements

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable to REPUBLIC PROTOCOL.

- ✓ There are migration scripts.
- ✓ There are tests.
- ✗ The tests are related to the migration scripts and a clear separation is made between the two.
- ✓ The tests are easy to run for CHAINSECURITY, using the documentation provided by REPUBLIC PROTOCOL.
- ✓ The test coverage is available or can be obtained easily.
- ✓ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ✓ The project only contains audit-related files, or, if not possible, a meaningful separation is made between modules that have to be audited and modules that CHAINSECURITY should assume correct and out of scope.
- ✓ There is no dead code.
- ✓ The code is well documented.
- ✓ The high-level specification is thorough and allow a quick understanding of the project without looking at the code.
- ✓ Both the code documentation and the high-level specification are up to date with respect to the code version CHAINSECURITY audits.
- ✓ There are no getter functions for public variables, or the reason why these getters are in the code is given.
- ✓ Function are grouped together according either to the Solidity guidelines⁴, or to their functionality.

⁴<https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions>

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.

Darknodes and darknode registry

1.1 Basic specification

- ✓ A node has to pay a 100,000 REN bond to register
 - | This does not hold and a corresponding issue can be found in section I.
- ✓ The registry does not require more than this bond to register
 - | This property holds: as long as the darknode pays the minimum bond it is immediately added to the list of darknodes. As a side note, there are other function that check whether the darknode is part of the current epoch or not yet.
- ✓ If a match cannot be verified to have been computed correctly by a node (FP), its bond is slashed and the node is deregistered (by the DarknodeSlasher)
 - | Functionality to challenge matched orders as well as slashing and deregistering a malicious nodes exist. Once a mismatch was confirmed and the appropriate function calls made, the node loses half its bond and is deregistered, under the assumption that the front-running attack described in 1.2 does not occur.
- ✗ Challengers win half of the bond if the challenge is won
 - | This does not hold and is elaborated in issue 1.2.
- ✗ Registry commits to a pod configuration for an epoch and then reshuffles
 - | This functionality is implemented off-chain and can not be verified. However, a LogNewEpoch event is emitted when an epoch interval has passed. Therefore, the registry is doing what is necessary to support that functionality.
- ✗ There cannot be more than 10,000 darknodes
 - | This cannot be verified as the bond value can be changed. This is further explained in I..
- ✓ A honest darknode can deregister and receive his bond back
 - | This only holds true under the assumption of a correct ownership chain. Otherwise a similar situation as in 1.2 arises.

1.2 Challengers might not receive rewards M ✓ Addressed

The process of dealing with the bond after slashing a malicious node allows for a scenario in which challengers might not receive their reward tokens:

- When a darknode registers in DarknodeRegistry his submitted bond gets transferred to the corresponding DarknodeRegistryStore. However, when the DarknodeSlasher calls the slash function of the DarknodeRegistry contract, the reward gets transferred to the receivers from the registry, not its store. Hence, no guarantees can be given that DarknodeRegistry has enough funds to pay out rewards. CHAINSECURITY is aware that in case of a correct ownership chain this is not critical, but since the ownership roles can be changed no strong guarantees can be given. Related design issue can be found in section I. and II..

Additionally, because it is necessary to submit two transactions (calls to submitChallengeOrder in the DarknodeSlasher contract), a malicious node can monitor if someone is about to challenge his matched order and front-run it with a deregister call. While he would also need to call or just wait for an epoch to pass, an attacker could have enough time to call refund and withdraw his bond before the call to slash is executed, leaving no bond to be slashed.

Likelihood: Medium

Impact: Medium

Addressed: The bond now directly gets transferred to the registry store. As to slash a bond the Darknode must have already been registered, its bond is guaranteed to be available to the Darknode registry given no change of the store ownership. Additionally, REPUBLIC PROTOCOL remarks that a 48-hour time period for challengers to submit the necessary contract calls is sufficient and notes that there is a cooling window of at least 48-hours for the Darknodes when deregistering.

Dark pools

2.1 Basic specification

- ✓ The public information is enough for traders to choose a pool which meets their needs in the near-future, including Signature requirements, Settlement layers, Incurred fees and Supported tokens.

General comment on public information: The bytecode of smart contracts is always public, but having the bytecode can not be considered equivalent with having a well-documented source code that verifiably matches the deployed bytecode. However, as it has become good common practice we assume the public availability of such source code. Furthermore, the source code should come with up-to-date documentation and ideally an independent audit describing the code's properties and guarantees.

Such public information has to be taken with a grain of salt, if, as in the case of RenEx, upgradable functionality exists. Then, the checks that were made ahead of time, do not necessarily hold at a later point. This applies to all of the following statements.

- Signature requirements: Based on code and documentation a trader can identify the requirements. In case of RenEx a trader can only withdraw within 48 hours with the signature of a broker.
- Settlement layers: Pools might offer different settlement layers with different functionalities. Depending on the trader's preferences, he can chose accordingly. Note, that these settlement layers can also differ in security guarantees as shown in the report below.
- Incurred fees (from broker services, order book listings): All fees are generally observable ahead of time. Hence, the trader can compare different pools.
- Supported tokens: As explained below, some pool designs might not support certain tokens, due to their design. Hence, the documentation should clearly state what kinds of tokens are supported.

- ✗ Darknodes that confirmed order matches and initiate settlement are guaranteed to be rewarded with a fee

For the Renex implementation we find the following: Darknodes that submitted the order using `submitOrder` will be rewarded once two orders have been successfully matched and settled. However, the darknodes which called `confirmOrder` and `settle` do not get any rewards.

Furthermore, other darknodes might try to front-run calls to `submitOrder` in order to receive the reward. Hence, the darknode that did the initial work and called `submitOrder` is not guaranteed to be rewarded.

Order book

3.1 Basic specification ✓ No Issue

- ✓ Once a match is found involved orders are updated to be confirmed
 - Once `confirmOrder` was successfully called, orders from both sides, buy and sell, are updated to `OrderState.Confirmed`.
- ✓ Each valid order has a defined state (open, closed, confirmed)
 - Orders are strictly typed as set by the `enum OrderState`. It consists of the four fields: Undefined, Open, Confirmed, Canceled. Upon order creation an order's state is set to be Undefined and afterwards can only be changed to any of the other three fields.
- ✓ Orderbook cannot confirm order matches that involve the same order, meaning that an order cannot be settled on multiple times

CHAINSECURITY remarks that parts of the functionality described above can be implemented in the settlement layer, which is why strong guarantees about the general case cannot be given.

However, the RenEx implementation correctly checks that only orders that are Submitted can be settled, updating the status to Settled afterwards and thereby preventing duplicate settling on any orders.

- ✓ The only allowed state transitions are: Undefined -> Open, Open -> Canceled, Open -> Confirmed

State transitions are only possible through the following functions:

- openOrder
- confirmOrder
- cancelOrder

These implement only the allowed transitions.

3.2 Unfixed DoS in orderbook ✓ Fixed

The cancelOrder function mentions a still unfixed Denial-of-Service attack:

```
121     function cancelOrder(bytes _signature, bytes32 _orderId) external {
122         if (orders[_orderId].state == OrderState.Open) {
123             // Recover trader address from the signature
124             bytes memory data = abi.encodePacked("RepublicProtocol:cancel:"
125                 , _orderId);
126             address trader = Utils.addr(data, _signature);
127             require(orders[_orderId].trader == trader, "invalid_signature");
128         } else {
129             // An unopened order can be canceled to ensure that it cannot be
130             // opened in the future.
131             // FIXME: This create the possibility of a DoS attack where a node
132             // or miner submits a cancelOrder with a higher fee everytime they
133             // see an openOrder from a particular trader. To solve this, order
134             // cancelations should be stored against a specific trader.
135             // Note: This is not a problem for 'openOrder', which will require
136             // broker signatures.
137             require(orders[_orderId].state == OrderState.Undefined, "invalid_
138                 order_state");
139         }
140         orders[_orderId].state = OrderState.Canceled;
141         orders[_orderId].blockNumber = block.number;
142     }
```

Orderbook.sol

A simple attack scenario in which an attacker monitors this contract for orders with OrderState.Undefined (e.g. by querying the getOrders function of the contract which is view and hence can be done for free) and sends a transaction invoking the cancelOrder function on this _orderId, is still possible.

Likelihood: Medium

Impact: Medium

Fixed: The cancelOrder function was changed. Now, an order can be cancelled by the trader who opened the order, or by the broker verifier contract. This allows the settlement layer to implement its own logic for cancelling orders without trader interaction and removes the possibility of the attack described above.

3.3 Silent failures ✓ Fixed

Opening orders with an invalid signature results in orders opened with trader address 0x0. This is the consequence of missing error handling and a silent failure as described in detail below.

As both are **external**, the functions openBuyOrder and openSellOrder in the Orderbook contract can be called by anyone. Both also subsequently invoke the private openOrder function.

It is possible to pass the **require** at the beginning of the function by paying the order opening fee and the trader address then gets recovered through `Utils.addr(data, _signature)`. This function uses `ECRecovery.recover()` to recover the traders address out of the signature. However it returns `0` if the recovery of the address fails⁵.

There is no check on what address has been returned, so the order opening will continue with `0x0` as the trader address, resulting in unfillable orders on the books that can be avoided.

Fixed: The functions `openSellOrder`, `openBuyOrder` were removed and the `openOrder` function now has **external** visibility. Also, signatures are now verified by the broker verifier contract of a settlement layer which needs to implement a given interface and provide a `verifyOpenSignature` function. This function call is additionally guarded by a **require**. Only the owner of a settlement registry can set the `BrokerVerifier`.

3.4 orderID might leak information

Order details are supposed to be invisible in the orderbook, however orders are stored as the hash of their details. The `orderId` is constructed as follows:

```
89     SettlementUtils.OrderDetails memory order = SettlementUtils.
      OrderDetails({
90         settlementID: _settlementID,
91         tokens: _tokens,
92         price: _price,
93         volume: _volume,
94         minimumVolume: _minimumVolume
95     });
96     bytes32 orderId = SettlementUtils.hashOrder(_prefix, order);
```

RenExSettlement.sol

For the security guarantees to hold, `orderId` has to represent a hiding commitment. However, everything except the `_prefix` can be known/guessed by an attacker, e.g. if he wants to check if a specific order exists. The hiding property therefore depends on entropy of `_prefix`. If it is randomly chosen and of sufficient length, the commitment can be considered hiding.

However, should it follow a specific format with a worst case being a predictable sequence like increasing `traderIds`, an attacker can easily guess those, allowing to check whether a specific order exists in the order book with reasonable computational effort.

Likelihood: Low

Impact: High

Addressed: REPUBLIC PROTOCOL remarks that the use of a 128-bit random nonce included in the order prefix prevents the hash from leaking information.

3.5 orderOpeningFee is lost in OrderBook

When `openOrder` is called in `OrderBook`, the `orderOpeningFee` has to be paid. This fee is transferred into the `OrderBook` contract, as seen below:

```
99     function openOrder(bytes _signature, bytes32 _orderId) private {
100         require(ren.transferFrom(msg.sender, this, orderOpeningFee), "fee transfer
      failed");
```

Orderbook.sol

However, the `OrderBook` contract does not contain any functionality to forward these tokens. Hence, they are permanently locked in `OrderBook`.

Likelihood: High

Impact: Medium

Fixed: Calls to `openOrder` of the `Orderbook` contract do not require fee payments anymore. Hence, the above issue is not relevant anymore and should fees be introduced or handled, this needs to be done through the settlement layer contracts.

⁵See <https://solidity.readthedocs.io/en/latest/units-and-global-variables.html#mathematical-and-cryptographic-functions>

Traders and brokers

4.1 Basic specification

- ✗ Can open and cancel orders on dark pools at any time, assuming the required broker signature for that dark pool has been acquired
 - CHAINSECURITY notes that a darkpool can theoretically prevent the opening of new orders by updating the `_orderOpeningFee` to a extremely high value. The owner is free to update the fee as he wants by calling `updateFee`.
 - Also, we note that the REN TOKEN is pausable. Hence if the REN TOKEN is paused, it will not possible to open new orders as the transfer of the fee will fail.
 - Cancelling an open order is always possible for a trader, given he can provide the appropriate order ID and signature.
- ✓ Once match found, orders are settled and funds swapped
 - This is dependent on the concrete settlement layer. However in the case of the RenEx reference implementation, the `settle` function is invoked after a match is found and the exchange of funds is initiated.
- ✓ Brokers can be involved to second-sign an order opening and demand a fee for their services (e.g. at every successful order opening)
 - The availability of this functionality is once again dependant on the concrete settlement layer, but is present in the reference implementation.

Miscellaneous

5.1 Basic specification

- ✓ No exhaustion of block gas limit by a function can block a trade
 - CHAINSECURITY investigated the behaviour of functions in terms of gas costs and could not identify a potential exhaustion of the block gas limit.
 - Average gas consumption on function calls as provided in the test suite by REPUBLIC PROTOCOL are an order of magnitude away from the limit.
 - There are no loops except in the `Utils` contract, which is inside a pure function and safe.
 - The linked list has no unbounded traversals.
- ✗ Once any information has been revealed, there is only one next state
 - This is not the case for a late call to `redeem` during the atomic swap process, as is described in the second suggestion at the end of this report.
 - Apart from this, while there are situations where after the revaluation of information two states are possible (e.g. `redeem` or `refund`) these are well-defined.

5.2 Unchecked arithmetic operations

REPUBLIC PROTOCOL does not make use of a secure library for arithmetic operation such as SafeMath and directly performs calculations without checks for under- or overflows. This situation leads to the fact that potential problems can occur in several spots:

- In the `subtractDarknodeFee` function of the `RenExSettlement` contract, an overflow will occur if `_value*(DARKNODE_FEES_DENOMINATOR-DARKNODE_FEES_NUMERATOR)` becomes bigger than $2^{256} - 1$. The function returns the new trade amount value and the fee in a struct. The consequence of this overflow will be that the new value to be returned will be higher than the original value.
Additionally, we note the rounding problem. This function should calculate the fee and a new value (being old value minus the fee). The fee is specified as 0.2%. However, if the value passed to the function is 2, then the rounding degrades the new value to 1, resulting in a 50% fee.

- In `RenExBalances`, function `transferBalanceWithFee`, two unchecked `uint` parameters are added, allowing an overflow. However, the function is only callable by the `RenExSettlement` contract.
- `RenExSettlement` contains two functions: `calculateSettlementDetails`, `calculateAtomicFees`. Both compute `orderDetails[_buyID].price + orderDetails[_sellID].price`, adding two `uint` price fields. As CHAINSECURITY understands, this may directly impact the midpoint price calculation, as each price could reach `uint_max/2`.
- In `DarknodeRegistryStore` a unchecked subtraction of two `uint` bond values happens. As these can be set arbitrarily by the owner of this contract, an underflow can occur.
- The `getOrders` functions in the `Orderbook` can be overflowed by passing a big offset and limit. While this only impacts the caller who as a result will not obtain all orders, this can be avoided.

Likelihood: Medium

Impact: Medium

Fixed: REPUBLIC PROTOCOL now homogeneously uses the `SafeMath` library.

RenEx Settlement and fee payment

6.1 Basic specification

- ✓ When performing atomic swaps traders don't need to reveal anything about their orders ahead of time
 - As at the time when an atomic swap is initiated the order is already matched, this holds. The details are revealed in the following order by interaction through the `RenExAtomicInfo` contract:
 - First, Bob submits the swap details which Alice or anyone knowing the `orderId` can observe. These include the redeemer address, amount of bitcoins and timestamp to redeem the atomic swap.
 - Second, if Alice audits and agrees, she initiates and submits her swap details. Now Bob or anyone knowing the `orderId` can check the same details.
- ✓ During an atomic swap traders are fully in control of their funds (until the settlement)
 - While actors are in control, they lock up their funds for a certain timeframe. This window is given by the specification, but the traders (or the dApp used to interact with the contract) sets this value when calling the respective function.
 - If Alice needs to lock up ether she calls `initiate` where the argument `timelock` specifies the time she locks up her ETH. Bob locks up his BTC for the trade by creating a Bitcoin script where he specifies the expiration time.
- ✓ By aborting an atomic swap there is a hard bound on the amount of time a counterparty's funds can be locked up
 - This holds. The trader locking up his funds sets the timeout himself, either via the `initiate` function of the `RenExAtomicSwapper` contract to lock up ETH or in the Bitcoin script locking up BTC. Hence a bound on the timeout they would have to wait to get their funds back if the counterpart fails to proceed with the swap exists.
 - CHAINSECURITY remarks that for the Bitcoin script this depends on the correct implementation of the script which is outside of the scope of this audit.
- ✗ A trader with an exchange account cannot open orders exceeding his balance
 - This is currently not enforced. CHAINSECURITY remarks that one can open orders from invalid addresses (as shown in 3.3). Additionally, it is possible to open exchange orders and then withdraw funds as described in 6.2.
- ✗ A trader's balance verification before an order submission does not reveal his balance or the price point when using an exchange account

This verification is done by the broker. The trader submits his details and the `orderId` and if the broker then agrees he sends the signature to the trader.

CHAINSECURITY remarks that off-chain and third party processes cannot be verified.

- ✗ A confirmed order match is automatically settled by the smart contract without interaction from the traders when using an exchange account

This does not happen automatically - orders are only confirmed by Darknodes calling `confirmOrder` in the `Orderbook` contract. Then a call to `submitOrder` and `settle` in the `RenExSettlement` module is needed.

CHAINSECURITY notes that there exists a clear incentive to do so since whoever called `submitOrder` will receive the fees.

- ✗ Traders pay a 0.2% fee in the Ethereum based token of the trade of which 80% are received by the darknodes

It should be noted that these numbers only refer to the settlement in the `RenEx` implementation. More so, it seems that the darknode currently receives 100% of the fee.

For both settlement options, through the `RenExSettlement` and `RenExAtomicSwapper` contracts, problems due to unchecked arithmetic operations in the call to `subtractDarknodeFee` can arise. Further elaboration can be found in the corresponding issue 5.2.

- ✓ The only allowed state transitions are: None -> Submitted, Submitted -> Settled, Settled -> Slashed

| With the scope limitation of this audit in mind, the above property holds.

6.2 Funds backing orders can be withdrawn ✓ Addressed

The following attack scenario is possible:

- An attacker creates an exchange order in the orderbook that is unlikely to be fulfilled, gets a broker to sign this (valid) order and opens it.
- Immediately afterwards he calls `signalBackupWithdraw` which would enable him to withdraw his funds 48h later. 48h later, he withdraws his funds but the order stays in the orderbook as `Open`.
- If the order gets matched at a later point, it gets removed from the orderbook by being marked as `OrderState.Confirmed` in the `Orderbook` contract.
- The `RenExSettlement` can begin: `submitOrder` can be called successfully. This sets the `orderStatus` to `Submitted`. CHAINSECURITY notes that `Orderbook.OrderState` is different from the `orderStatus` in `RenExSettlement`.
- The subsequent `settle()` function however will not be able to complete successfully as `execute()` will fail due to insufficient funds in the trader's balance. The orders can't get unmatched anymore, so the other trader's order (the honest one) is also stuck.

This could be used to set up a bunch of useless orders which would then inhibit trading once the price moves in this range. It seems like the broker or matching node must make sure that all open orders are cancelled if the trader `signalsBackupWithdraw`, since it is not enough to just stop signing new orders and hoping the already opened orders will be executed before the 48h are over.

Likelihood: Medium

Impact: Medium

Addressed: REPUBLIC PROTOCOL updated the specification to discuss, in more detail, the 24-hour expiry limit for orders and thus considers the 48-hour withdrawal time sufficient to prevent this being an issue.

6.3 Double-spending with RenEx atomic swaps

The way the protocol is currently described⁶, it allows for double-spending during the atomic swap. Given that the atomic swap spans a time period of 72 hours, it is not unlikely that two traders get matched twice during this time frame. In case this happens, either trader can double-spend during the atomic swap.

For the simplicity of the argument we assume that the two matched orders have identical value and that the bitcoin holder (Bob) is acting malicious. Bob can execute his attack as follows:

1. Bob follows the protocol, but for both orders Bob transmits information about the same bitcoin transaction.
2. On both orders, Alice will perform the checks and finds that all the parameters are correct.
3. Alice will initiate both swaps and share the two different swap details.
4. Bob audits both swaps and if correct, redeems both swaps.
5. Alice tries to redeem two bitcoin transactions, but finds that they are in fact one transaction.

Note, that this can be extended with more parallel orders and that a malicious trader can behave such that this scenario is very likely to occur.

In order to protect from this, Alice needs to verify that both of Bob's submitted `_swapDetails` do not point to the same timelocked bitcoin transaction when auditing the details. Accordingly, Bob needs to monitor that there is more than one `LogClose` event (otherwise Alice is trying to double-spend on him).

Likelihood: Low

Impact: High

Fixed: The unique `swapID` is now deterministically generated from the secret hash and the timelock, preventing such behaviour.

6.4 Malicious trader can defer call to redeem

The final on-chain transaction of a successful atomic swap is a call to `redeem()`. However, the `redeem()` function does not check whether the swap has already expired (i.e. if the timelock has passed):

```
181         SettlementUtils.OrderDetails memory order = SettlementUtils.  
            OrderDetails({  
182             settlementID: _settlementID,  
183             tokens: _tokens,  
184             price: _price,  
185             volume: _volume,  
186             minimumVolume: _minimumVolume  
187         });  
188         bytes32 orderID = SettlementUtils.hashOrder(_prefix, order);
```

RenExAtomicSwapper.sol

Therefore, a malicious trader can defer its call to `redeem()`, as it can still be called successfully once the swap is already expired. Out of this, two cases arise:

- The other trader calls `refund()` before the bitcoin script expires: In this case the malicious trader front-runs with a call to `redeem()`. Now, the malicious trader received its funds, while the other trader only has limited to retrieve the bitcoins. In case the honest trader fails to do this in time, the malicious trader obtains both funds.
- The other trader doesn't call `refund()` before the bitcoin script expires: The malicious trader automatically is refunded the bitcoins and can claim the ETH through `refund()`. Therefore, the malicious trader has obtained both funds.

Likelihood: High

Impact: Medium

Fixed: REPUBLIC PROTOCOL updated the specification to discuss the timing requirements of traders and the symmetric nature of the atomic swapping.

⁶<https://github.com/republicprotocol/renex-sol/blob/master/docs/03-atomic-swapping.md>

6.5 Requirements for Used Tokens ✓ Acknowledged

REPUBLIC PROTOCOL relies on the correctness of the used tokens for the correctness of their system. Any malicious tokens, where token balances can be manipulated or where ERC20 functions are incorrectly implemented would jeopardize the funds of traders and darknodes.

Furthermore additional care is required when dealing with the following token features:

- Inflationary/Deflationary Tokens (i.e. the balance can change without additional transfers): Deflationary tokens would currently break the logic as the internal balance would not be updated accordingly.
- Tokens with transfer fees: If fees occur for token transfers, REPUBLIC PROTOCOL might not have sufficiently many tokens during withdrawal.
- Built-in locking/vesting schemes: malicious actors might be able to block the slashing process as their tokens cannot be withdrawn and hence provide no incentive for slashing.

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 REPUBLIC PROTOCOL, including in REPUBLIC PROTOCOL's ability to deal with such powers appropriately.

Bond of darknode can be abused

By calling the function `updateDarknodeBond` the owner of `DarknodeRegistryStore` can set the value of the registered bond of any darknode to an arbitrary `uint256` parameter.

This function is used by the `slasher` role of the `DarknodeRegistry` contract after a darknode has been found a failed prover and his bond is to be slashed in half. The challengers' reward is then payed out directly (and not added to their bond) with a different function, by transferring `REN TOKENS`.

There seems to be no use case where the value of a bond is increased without depositing the according `REN TOKENS`. This is potentially dangerous, considering that at one point one may want to withdraw these.

Hence, in the case of a malicious or colluding admin it is possible to increase a darknode's bond to the amount of `RENs` available in this contract (i.e. its total balance) and then this cooperating node can withdraw it after deregistering again and calling `refund`. Note, that the malicious admin can only call this function right after deployment as the ownership has to be transferred afterwards.

Addressed: Darknode bonds can now only be decreased, not increased, through the `updateDarknodeBond` function. Additionally, ownership of the store contract can be claimed by the `DarknodeRegistry` by calling the `claimStoreOwnership` method.

All atomic swaps can be slashed

Through the `slash` function inside `RenExSettlement` any atomic swap can be slashed by the designated `slasherAddress`. This functionality is intended to punish traders that fail to complete the atomic swap.

In contrast to the darknode slashing, which is evidence-based, this slashing is possible for all settled orders with atomic swap settlement. It does not check whether the atomic swap actually failed.

However, given that the damage is fairly low (0.4% of trade value) and that the misbehaviour is clearly observable, this issue has a low risk.

Acknowledged: REPUBLIC PROTOCOL notes: This will be altered in future versions of `RenEx` where governance is not controlled by the REPUBLIC PROTOCOL team, but is controlled by a more decentralized system (e.g. a relay, or a voting mechanism).

`submissionGasPriceLimit` can selectively freeze functionality

Theoretically, REPUBLIC PROTOCOL can use its power of setting the `submissionGasPriceLimit` to selectively block certain orders or traders. In this case, REPUBLIC PROTOCOL would front-run undesired calls to `submitOrder` with calls to `updateSubmissionGasPriceLimit` so that it would make the order creation fail.

However, this is highly unlikely and comes with relatively little impact.

Fixed: The submission gas price limit cannot be lower than 0.1 gwei anymore, restricting such behaviour.

Owner of `RenExSettlement` can selectively manipulate

Theoretically, the owner of the `RenExSettlement` can use its power of setting the state variables, such as `renExBalancesContract`, to influence the state to the advantage or disadvantage of certain traders. As an example, once an call to `slash` is observed for such a user, the owner can set `renExBalancesContract` to `address(0)` using the `updateRenExBalances` function. Then, the call to slashing `transferBalanceWithFee` has no effect but also does not lead to a revert. Afterwards, the `renExBalancesContract` variable can be returned to the old state.

Overall, the owner of the RenExSettlement contract (and similarly the owner of the RenExBalances contract) is in a very powerful position, which could lead to abuse. However, this seems fairly unlikely given the directly observability.

Acknowledged: REPUBLIC PROTOCOL remarks that After observing decentralized governance in action in other projects, a DAO will designed and implemented and will take ownership of all Republic Protocol and RenEx contracts.

Design Issues

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

Inefficient transfers

The DarknodeRegistry contract receives a bond from a Darknode upon its registration and immediately sends the funds further, unguarded:

```
function register(address _darknodeID, bytes _publicKey, uint256 _bond)
    external onlyRefunded(_darknodeID) {
    require(_bond >= minimumBond, "insufficient_bond");
    require(ren.transferFrom(msg.sender, address(this), _bond), "bond_
        trasfer_failed");
    ren.transfer(address(store), _bond);
```

DarknodeRegistry.sol

The two transfers can be easily combined into a single one which should be guarded by a **require**. This issue is related to item 1.2.

Fixed: REPUBLIC PROTOCOL changed this to a single transferFrom call which is guarded by a **require**.

Missing requires

The DarknodeRegistry contract does not guard token transfers that are supposed to reward challengers with **requires**:

```
// Reward the challengers with less than the penalty so that it is not
// worth challenging yourself
ren.transfer(store.darknodeOwner(_challenger1), reward);
ren.transfer(store.darknodeOwner(_challenger2), reward);
```

DarknodeRegistry.sol

This issue is related to item 1.2 which is a consequence of the missing checks.

Fixed: The transfers are now within **require** checks.

Simple storage savings possible

Reordering the storage of the RenExSettlement variables can save a storage slot by e.g. moving the uint32 fields down and close to the int16 fields:

```
// This contract handles the settlements with ID 1 and 2.
uint32 constant public RENEX_SETTLEMENT_ID = 1;
uint32 constant public RENEX_ATOMIC_SETTLEMENT_ID = 2;

// Fees in RenEx are 0.2%. To represent this as integers, it is broken
// into
// a numerator and denominator.
uint256 constant public DARKNODE_FEES_NUMERATOR = 2;
uint256 constant public DARKNODE_FEES_DENOMINATOR = 1000;

// Constants used in the price / volume inputs.
int16 constant private PRICE_OFFSET = 12;
```

```

int16 constant private VOLUME_OFFSET = 12;

// Constructor parameters, updatable by the owner
Orderbook public orderbookContract;
RenExTokens public renExTokensContract;
RenExBalances public renExBalancesContract;
address public slasherAddress;
uint256 public submissionGasPriceLimit;
}

```

RenExSettlement.sol

CHAINSECURITY expects that more such simple optimizations can be considered in the REPUBLIC PROTOCOL contracts.

Fixed: The savings were implemented by REPUBLIC PROTOCOL.

Suboptimal struct swap

Gas savings can be introduced by reordering the **struct** Swap in the RenExAtomicSwapper contract, if the struct is declared as follows ():

```

struct Swap {
    uint256 timelock;
    uint256 value;
    bytes32 secretLock;
    bytes32 secretKey;
    address ethTrader;
    address withdrawTrader;
}

```

RenExAtomicSwapper.sol

Roughly a hundred gas are saved in transaction costs and an additional hundred in execution costs by placing the address fields last. Considering that swaps will be frequently used throughout the lifetime of RenEx, the compounded savings will be significant.

Fixed: REPUBLIC PROTOCOL declared the struct as recommended.

Redundant casts

The DarknodeRewardVault introduces an unnecessary cast in its withdraw function. There, the parameter **address** _darknode is passed and then again cast to the **address** type.

```

function withdraw(address _darknode, ERC20 _token) public {
    address darknodeOwner
        = darknodeRegistry.getDarknodeOwner(address(_darknode));
}

```

DarknodeRewardVault.sol

Another occurrence of such a redundant cast can be found in the onlyDarknode modifier of the Orderbook contract.

Fixed: The redundant casts were removed.

deposit breaks Checks-Effects-Interactions pattern

The deposit function in the DarknodeRewardVault contract is breaking the Checks-Effects-Interactions pattern⁷. In this particular case it is not a critical issue, since:

⁷ <https://solidity.readthedocs.io/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern>

- A modifier allows only the settlement contract to call the function, so the `DarknodeRewardVault` whose function is externally called by `RenExBalances` in the reference implementation cannot do that.
- The `DarknodeRewardVault` contract to which the fees are transferred is specified by the owner of the settlement contract. Hence the responsibility lies with the owner to inspect the `DarknodeRewardVault` it will be using.

Nonetheless, an external call is happening before local state changes are executed which goes against best practices and should be avoided. Additionally, further restrictions to increase security can be made by limiting the gas amount `RenExBalances` makes available for the external call, e.g.:

```
rewardVaultContract.deposit.value(_fee).gas(<gas units>)
    (_feePayee, ERC20(_token), _fee)
```

Acknowledged: REPUBLIC PROTOCOL acknowledges this and regards it as safer to keep this function in its current form than to change it.

Outdated compiler version

A new version of the solidity compiler, 0.4.25, was recently released. The new release introduces several important fixes⁸ and REPUBLIC PROTOCOL should make use of the newest compiler version unless there is an explicit and documented reason not to do so.

Fixed: The compiler version was upgraded in all contracts.

Atomic swaps generate free options

During an atomic swap as implemented in the `RenExAtomicSwapper` contract, the Bitcoin holder has the power to wait 24 hours before closing the swap using a call to `redeem()`. During this time, the Bitcoin holder can observe the price development which essentially boils down to a free put option on Bitcoin that can be settled at any time.

It is expected that traders will use this, given that the threat model of REPUBLIC PROTOCOL expects rational actors and considering the high volatility of cryptocurrency markets which allows significant earnings using this strategy.

Concretely, a swap of 10 BTC to 330 ETH might be matched initially implying an exchange rate at the start of roughly 33 BTC/ETH. If in the following 24h the rate decreased, the Bitcoin owner will fulfill the swap and exercise the option. In case the trader does not complete the swap, it will be slashed for 0.4% of the trade value. Therefore, a rational trader will complete the swap if the rate increased by less than 0.4% and otherwise accept the slashing.

As a consequence of this observation, ETH traders must always factor in a premium when setting rates.

Acknowledged: REPUBLIC PROTOCOL remarks that the introduction and implementation of the KYC and blacklisting process now ensures that this issue can only occur once by a trader. The proposed future improvement to the specification of REPUBLIC PROTOCOL/ RenEx will continue to address this issue.

Better definition of “sufficient time” for atomic swap

The documentation⁹ states that Alice (who is converting ETH to BTC) needs to ensure that “she has enough time to redeem the atomic swap.” CHAINSECURITY recommends that this condition should be more clearly defined to avoid integration and usage issues.

This time has to be evaluated before initiating the swap on-chain. It is important to note, that it is not sufficient to check for a large remaining time in the bitcoin script (e.g. 23 hours) which would be easily sufficient to withdraw the BTCs. The time to be checked is: (deadline of the bitcoin script - **now**) - (swap.timelock - **now**)¹⁰. This is length of Alice’s guaranteed redemption phase. If the length of this time is very small, Alice might not be able to redeem the BTCs and lose her funds. If the time is negative, Bob can delay his call to `redeem()` such that he can obtain both BTC and ETH and Alice loses all of her funds.

⁸<https://blog.ethereum.org/2018/09/13/solidity-bugfix-release/>

⁹<https://github.com/republicprotocol/renex-sol/blob/master/docs/03-atomic-swapping.md>

¹⁰Note, that according to REPUBLIC PROTOCOL’s suggestions (swap.timelock - **now**) will generally be 24 hours at that moment.

Addressed: The specification has been updated to discuss timeframes more explicitly.

RenExAtomicInfo **should emit more events**

The atomic swap documentation¹¹ mentions several cases where actions are triggered by a previous transaction. As an example, Bob submits swap details using the `submitDetails` function which Alice then has to query. However, `submitDetails` does not emit an event to alert Alice and hence Alice needs to constantly check by making calls to `swapDetails`. This workflow could be simplified with events.

Fixed: This contract is no longer used.

Possible gas optimizations in RenExAtomicSwapper

The RenExAtomicSwapper contract uses a **struct** to store information about the swaps:

```
6      struct Swap {
7          uint256 timelock;
8          uint256 value;
9          address ethTrader;
10         address withdrawTrader;
11         bytes32 secretLock;
12         bytes32 secretKey;
13     }
```

RenExAtomicSwapper.sol

Some function, including `redeem()` and `refund()`, completely load the **struct** into memory at the beginning:

```
106     function refund(bytes32 _swapID) external onlyOpenSwaps(_swapID)
107         onlyExpirableSwaps(_swapID) {
108         // Expire the swap.
109         Swap memory swap = swaps[_swapID];
109         swapStates[_swapID] = States.EXPIRED;
110
111         // Transfer the ETH value from this contract back to the ETH trader.
112         swap.ethTrader.transfer(swap.value);
113
114         // Logs expire event
115         emit LogExpire(_swapID);
116     }
```

RenExAtomicSwapper.sol

However, as seen above only two values of the **struct** are accessed. Hence, additional gas costs occur, because the remaining values were also loaded into memory. Gas costs could be reduced by accessing the required elements directly.

Fixed: REPUBLIC PROTOCOL implemented the proposed changes.

No Input Validation for `submissionGasPriceLimit`

The RenExSettlement contract has a `updateSubmissionGasPriceLimit` function to update the transaction gas price that can be used during calls to `submitOrder`. Given that a very low limit can freeze important functionality and given that this function will be called rarely, it would make sense to add a basic consistency check, e.g. `_newSubmissionGasPriceLimit > 108`.

Fixed: REPUBLIC PROTOCOL implemented the proposed changes.

¹¹<https://github.com/republicprotocol/renex-sol/blob/master/docs/03-atomic-swapping.md>

No Input Validation for Updates

The RenEx contracts contain multiple options to update their topology, such as the functions `updateOrderbook`, `updateRenExTokens` or `updateRenExBalances`. Furthermore, ownership can be transferred for some of them.

These updates are extremely critical and even short times of incorrect settings can lead to loss of funds. Hence, for contract upgrades functions could optionally perform such basic checks (as ensuring the address represents a contract or ensuring a consistent topology). For ownership updates, multi-step approaches can be used to avoid accidental loss of control.

Fixed: REPUBLIC PROTOCOL solved the problem as now two stages are required in ownership changes and a basic existence check for a `VERSION` tag is done when updated links between contracts.

Broker signature includes limited data

The broker has to provide a signature in order to allow a fast withdrawal from the `RenExBalances` contract. However, the `verifyWithdrawSignature` function only takes the trader's address and not the address of the withdrawn token as input. Consequently, the brokers signature meant for the withdrawal of one token could be used for the withdrawal of another token by the same trader.

Fixed: The `verifyWithdrawSignature` function now also takes the token address as a parameter.

Options creatable out of `RenExSettlement`

This issue is related to previous issue about options in atomic swaps. However, this issue is concerned with the standard RenEx settlement. A malicious trader *A* could create an option (on a Token \rightarrow Token trade¹²) as follows:

1. Trader *A* has 9 ETH in its `RenExBalance`.
2. *A* creates an order for 10 ETH \rightarrow 100 Tokens (`openOrder` in `Orderbook`) and waits for it to be confirmed (using `confirmOrder`). Note, that confirming this order is currently not prevented and will not be punished.
3. Then, *A* can submit the order through `submitOrder` in `RenExSettlement`.
4. At this point, anyone can call `settle`, however, it cannot be called successfully, because the execution will fail due to insufficient funds.
5. Now *A* waits for the right time, when it is beneficial to exercise its option.
6. To exercise, *A* deposits one additional ETH and calls `settle` to exercise the option.

Some of the steps above are not supposed to be possible according to the documentation, but the code currently permits them. The only cost for this option is the `orderOpeningFee` paid when opening an order.

Acknowledged: REPUBLIC PROTOCOL remarks that the introduction and implementation of the KYC and blacklisting process now ensures that this issue can only occur once by a trader. The proposed future improvement to the specification of REPUBLIC PROTOCOL/ RenEx will continue to address this issue.

¹²ETH is also considered a token in this context

Recommendations / Suggestions

- ✗ The naming of the modifier `onlyExpirableSwaps` is confusing, since it actually enforces that the timelock of the swap has already expired, `require(now >= swaps[_swapID].timelock)`. Hence it could be renamed to `onlyTimedOutSwaps` and described with "swap timed-out" instead of the less clear description "swap not expirable".
- ✓ REPUBLIC PROTOCOL should warn the users that a late call to `redeem()` in case of an atomic swap can lead to a loss of all funds. If the redemption transaction is not accepted within the blockchain in time, the trader does not receive the ETH, and also disclosed the secret to unlock the BTC, which will therefore likely be withdrawn.
- ✗ REPUBLIC PROTOCOL could rename the `ordersCount` function of the `OrderBook` to something like `historicalOrderCount`, given that the current comment:

```
/// @notice returns the number of orders in the orderbook
```

Orderbook.sol

might be misunderstood as the number of open orders.

- ✓ In the `RenExSettlement` the following comment mentions "non-priority" tokens:

```
/// @param _price The price of the order. Interpreted as the cost for 1  
/// standard unit of the non-priority token, in 1e12 (i.e.  
/// PRICE_OFFSET) units of the priority token).
```

RenExSettlement.sol

In other parts of the code, they are called "secondary" tokens. If these terms mean the same it would make sense to unify the terminology to avoid misunderstandings.

Post-audit comment: REPUBLIC PROTOCOL has fixed some of the issues above and is aware of all the implications of those points which were not addressed. Given this awareness, REPUBLIC PROTOCOL has to perform no more code changes with regards to these recommendations.

Disclaimer

UPON REQUEST BY REPUBLIC PROTOCOL, 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..