Smart Contract Security Audit Report

Loopring Protocol Smart Contract version 2

SECBIT

Dec 15, 2018

1. Introduction

Loopring Protocol Smart Contract version 2 is a set of smart contracts in the Loopring ecosystem that check order-rings from ring-miners, trustlessly settle and transfer tokens on behalf of users, incentivize ring-miners and wallets with fees, and emit events.

SECBIT Labs carried out an audit for Loopring Protocol Smart Contract version 2 from Oct 15, 2018 to Dec 15, 2018.

In the rest of this report, if not explicitly specified, **LPSC** stands for Loopring Protocol Smart Contract version 2, and **Loopring protocol** stands for Loopring protocol version 2.

1.1 Basic Information of LPSC

- **Project Website**
  - https://loopring.org/
- **Project Whitepaper**
  - https://github.com/Loopring/whitepaper
- **Audited Code**
  - https://github.com/Loopring/protocol2/tree/audit-1012
  - Commit `10100aa616223439516c48f2c76ef386e8f996ff`

According to the Loopring whitepaper, LPSC can be deployed on multiple types of blockchains. This audit focuses *only* on the version targeted for Ethereum.

1.2 Audit Process

The audit is processed around the formal model of LPSC in following steps.

1. Define the formal model of LPSC from LPSC implementation, which is used as the foundation of subsequent steps.
2. Manually review the consistency between the formal model and the LPSC implementation as well as the Loopring whitepaper, in order to discover obvious issues.
3. Attempt to formally prove some properties of LPSC upon the formal model, in order to show that LPSC is well-behaved. The disproving of a property usually implies bugs in the contract implementation.

Above formal model and property proofs are implemented in an interactive proof assistant **Coq**, and can be found at

- https://github.com/sec-bit/loopring-protocol2-verification

1.3 Issue List

Several issues have been discovered in the audit and are briefly listed below. They have been fixed in the latest version of LPSC, or have been addressed out of LPSC, or only have negligible effects in practice. Details of each issue can be found in the section marked in the list. The explanation of severity can be found in the appendix at the end of this report.

#	Type	Description	Severity	Status	Section
1	Impl Error	Mismatched tokens in adjacent orders	High	Fixed	4.1
2	Impl Error	withdraw() does not handle external call failures properly	High	Fixed	4.2
3	Impl Error	Incomplete check for multiple all-or-none orders in multiple rings	Medium	Fixed	4.3
4	Potential Risk	Potential GAS attack to ring-mines from order brokers/owners	Low	Impossible in practice	4.4
5	Potential Risk	High GAS consumption from abnormal token contracts	Low	Protected by relays	4.5
6	Potential Risk	Inaccurate filled amounts caused by rounding errors	Low	Loss too tiny in practice	4.6

The rest of this report is organized as below.

- Section 2 analyzes LPSC along with the formal model.
- Section 3 list LPSC properties checked by formal proofs.
- Section 4 shows details of issues discovered in the audit.
- Section 5 highlights best practices in LPSC.
- Section 6 concludes the quality of LPSC.

2. Contract Analysis

2.1 LPSC Overview

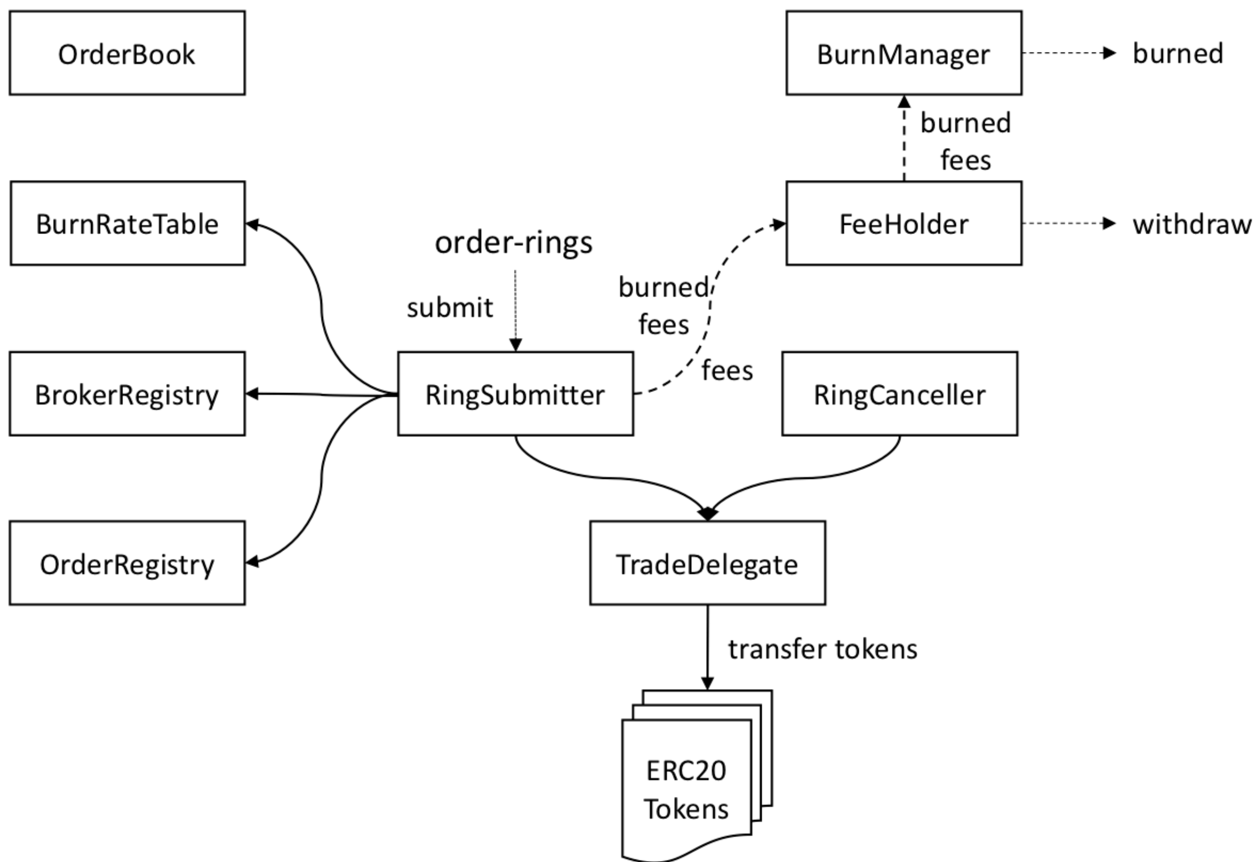
Loopring orders are organized in **order-rings**. Every order in a ring sells tokens to its immediate predecessor and buys tokens from its immediate successor. A ring must contain 2 ~ 8 orders in the current implementation. Each order basically records the following information:

- who owns the selling tokens,
- which token is sold and how much of it is desired,
- which token is bought and how much of it is desired,
- parameters to calculate fees each participation should get,
- broker of the order.

Order-rings can be produced by **ring-miners** in the **relay-mesh** network to which users submit their orders, or be produced by users who directly match their orders. The latter orders are called **P2P orders**.

In order to incentivize every participation, an amount of fees may be reduced from exchanged tokens and paid to ring-miners and wallets. According to which token the fees are paid in, a publicly observable percentage, *i.e.,* the **burn rate**, of miner fees are burned. In addition, LPSC supports ring-miners to waive a percentage of fees for specified orders, and even pay a percentage of fees from other orders to specified orders.

LPSC is deployed in multiple contracts (as shown in the following figure), which separate the functionality of LPSC.



- `RingSubmitter` accepts order-rings from users, validates rings and orders, calculates fees and fills, and makes the final transfer of tokens and fees. The token transfer between orders is made by calling `transferFrom()` of corresponding token contracts. Fees, including burned fees, are first recorded in `FeeHolder` contract of LPSC, which can be withdrawn later by fee recipients.
- `RingCanceller` provides a set of functions to cancel orders.
- `TradeDelegate` provides multiple functionalities:
    - a safe mechanism for `RingSubmitter` to interact with token contracts (*e.g.,* call `transferFrom()` of ERC20 token contract),
    - an interface for `RingCanceller` to accomplish the cancellation jobs,
    - a function for `RingSubmitter` to record the filled amounts of orders,
    - a query function to check the filled amounts and cancellation status of orders.
- `FeeHolder` is used by `RingSubmitter` to record and hold fees for each fee recipient, and provides functions for fee recipients to actually get the fee tokens in the corresponding token contracts.
- `BurnManager` provides a public function for everyone to actually burn the burned fees held by `FeeHolder`.

- `BurnRateTable` manages the burn rates of each token. A burn rate query function is provided for `RingSubmitter` and others to get the current burn rates of specified tokens. It also implements a mechanism to adjust the burn rates of each token.
- `BrokerRegistry` provides functions to query/register/unregister order brokers. The query interface is used by `RingSubmitter` in the order validation.
- `OrderBook` provides a public order book where order owners and brokers can save their orders on chain. Everyone can look for the order details in the order book by the order hash.
- `OrderRegistry` allows users to directly register their orders in LPSC. The further order signature verification in `RingSubmitter` is not required for those registered orders.

2.2 Formal Model

The audit is primarily carried out around a formal model of LPSC. The formal model, built from the LPSC implementation, abstracts and expresses behaviors of the LPSC implementation in a formal language, which is then used for the manual review and the formal proofs of properties of LPSC.

Both the formal model and the formal proofs are implemented in an interactive proof assistant **Coq**. Coq has been widely used in both academical an industrial projects for security guarantee and enhancement, such as operating systems, compilers and cryptography libraries. The underlying type and logic system in Coq can guarantee the correctness of proofs.

The Coq code of formal model and proofs of LPSC can be found at https://github.com/sec-bit/loopring-protocol2-verification. *This report only explains the overall framework and some special parts of the formal model and proofs, rather than goes deep into every detail.*

The formal model defines an abstract state machine for LPSC.

- The **machine state**, or **world state**, abstracts storage variables of LPSC and token contracts. A world state can be thought as a tuple, of which each element represents the semantics of one or multiple storage variables of the contract. For example, a storage variable of type `mapping(address => bool)` can be represented by a total map whose domain is the set of natural numbers and range is a set of boolean values.
- The **state transition** describes how the world state is changed, which events are emitted, and what value is returned on a successful message call to LPSC or token

contracts. State transitions only include the execution of message calls without revert. The revert of a message call in certain conditions (*e.g.,* a specific combination of argument values and storage variable values) is represented as the non-existence of state transition in such conditions.

At the top level of the state machine definition:

- The world state is defined by `WorldState` (in Coq file `Models/States.v`) as the aggregation of **sub-states** of LPSC contracts, token contracts and the blockchain (fields `wst_*` in `WorldState`).
- All possible message calls are defined by an inductive type `Message` (in Coq file `Models/Messages.v`), which aggregates all public/external functions of LPSC and token contracts.
- All events are defined by an inductive type `Event` (in Coq file `Models/Events.v`), which aggregates all possible events from LPSC and some pseudo events that aid the modeling and proving.
- The state transition is defined inductively by `lr_steps` (in Coq file `Models/TopModel.v`).
  - The base case `lr_steps_nil` of `lr_steps` says nothing changes if no message is called.
  - The induction case `lr_steps_cons` of `lr_steps` says the effects of the successful execution of one or multiple message calls are the concatenation of the sequential executions of them from the beginning to the end.

  For example,

  ```
  lr_steps wst (msg0::msg1::nil) wst' v (evt0::evt1::evt2::nil)
  ```

  means the successful execution of a sequence of message calls to `msg0` and `msg1` will change the beginning world state `wst` to `wst'`, return some value `v`, and generate a sequence of events `evt0`, `evt1` and `evt2`.

At the lower level, the state transition of the entire world state is divided into multiple contract-specific transitions. The contract-specific transition for a contract is composed of a set of **specifications** of public/external functions of that contract. A function specification models following three aspects of the function (as defined by `FSpec` in Coq file `Libs/LibModel.v`):

- `fspec_require` describes the requirements that must be satisfied in order to execute the function successfully without revert;
- `fspec_trans` describes how the successful execution of the function changes the

world state and what value is returned;
- `fspec_events` describes the events emitted from the successful execution of the function.

The rest of this section analyzes each LPSC contract along with its sub-state and function specifications.

2.3 RingSubmitter

`RingSubmitter` is the primary contract of LPSC. Its single public function `submitRings(bytes data)` is called by ring-miners and users to submit order-rings. `submitRings()` then calculates fills and fees for each order, and transfers tokens and fees at the end.

- Modeled contract files
  - `contracts/impl/RingSubmitter.sol`
  - `contracts/impl/Data.sol`
  - `contracts/helper/{Mining, Order, Participation, Ring}Helper.sol`
- Coq files
  - `Models/{RingSubmitter, States, Events, Messages}.v`

2.3.1 Sub-state

The sub-state of `RingSubmitter` is represented by `wst_ring_submitter_state` in `WorldState` and defined by `RingSubmitterState` in Coq file `States.v`. Each field of the sub-state represents a storage variable of `RingSubmitter` of the similar name (*e.g.,* `submitter_lrcTokenAddress` for storage variable `lrcTokenAddress`).

Because the primary portion of `submitRings()` operates memory variables rather than storage variables, the formal model of `RingSubmitter` in addition models them by an internal state `RingSubmitterRuntimeState` (in Coq file `RingSubmitter.v`) with following fields:

- `submitter_rt_mining`, defined by `MiningRuntimeState`, models the mining portion deserialized from `submitRings()` argument and its subsequent changes in memory;
- `submitter_rt_orders`, defined as a list of `OrderRuntimeState`, models the `Order` array deserialized from `submitRings()` argument and its subsequent changes in memory;
- `submitter_rt_rings`, defined as a list of `RingRuntimeState`, models the `Ring`

array deserialized from `submitRings()` argument and its subsequent changes in memory;
- `submitter_rt_token_spendables` is defined as a total map in Coq from the token owner and the token address to the corresponding spendable amount of `TradeDelegate`;
- `submitter_rt_broker_spendables` is defined as a total map in Coq from the token broker, the token owner and the token address to the corresponding spendable amount of the broker, which and the above `submitter_rt_token_spendables` are used to model various spendable amount calculations in `submitRings()` and its sub-calls.

2.3.2 Specifications

The specification of `submitRings()` is defined by `submitRings_spec` in Coq file `RingSubmitter.v`.

The byte array argument of `submitRings` is modeled by its semantics after deserialization, *i.e.,* a list of orders, a list of rings, and the mining parameters.

`submitRings` along with its sub-calls is large and complicated, so `submitRings_spec` models it by a sequence of **sub-specifications**, each of which models a small portion of `submitRings`.

- `update_orders_hashes_subspec`, `update_orders_brokers_and_interceptors_subspec`, `get_filled_and_check_canceled_subspec` and `check_orders_subspec` respectively model the update and check of various parameters of orders, including order hash, broker interceptors, filling/cancellation status, remaining spendable tokens/fees;
- `update_rings_hash_subspec` models the update of hashes of rings;
- `update_mining_hash_subspec` and `update_miner_interceptor_subspec` respectively model the update and check of various parameters about ring-miner;
- `check_miner_signature_subspec` and `check_orders_dual_sig_subspec` respectively model the check of order/ring/miner hashes and signatures;
- `calc_fills_and_fees_subspec` models the calculation of various fill amounts and fees;
- `validate_AllOrNone_subspec` models the validation of all or none status;
- `calc_and_make_payments_subspec` models the make of final payments of tokens and fees.

For such a complicated implementation, it specification and sub-specifications are mostly defined by following the Solidity code in Coq. For example, `RingHelper::checkOrdersValid()` called by `submitRings()` is modeled in Coq as below. Check comments in `(* *)` for the explanation.

(**Note**: There was a bug in `checkOrdersValid()` as mentioned in Section 4.1. Here we refer to the fixed version.)

```
(* Model the loop in checkOrdersValid() inductively.

   '_ring_orders_valid orders pps ps' means orders referred by
   participations 'ps' are valid.

   * 'pps' represents participations that have been iterated,
   * 'ps' represents the rest participations in the ring,
   * 'orders' represents all orders submitted to 'submitRings()'.

   If 'ps' represents all participations in a ring, then
   '_rings_orders_valid orders nil ps' represents the ring is valid.
*)
Inductive _ring_orders_valid
          (orders: list OrderRuntimeState)
          (pps: list Participation)
: list Participation -> Prop :=
| RingOrdersValid_nil:
  (* Base case: an empty ring (nil) is always valid *)
    _ring_orders_valid orders pps nil

| RingOrdersValid_cons:
  (* Inductive case:

     Suppose,
       * 'p' is the current participation being checked
       * 'p_ord' is the order referred by 'p'
       * 'pp' is the previous participation of 'p'
       * 'pp_ord' is the order referred by 'pp'
       * 'pps' are all checked participations
       * 'ps' are remaining participations (may be nil) except 'p'

     If
       * the valid field of the current participation 'p' is true,
       * the selling token of the current order 'p_ord' and the
         buying token of the previous order 'pp_ord' match, and
       * orders referred by remaining participations 'ps' are valid,
     then it can conclude orders referred by participations 'p::ps'
```

```
      are valid.
    *)
      forall p ps pp p_ord pp_ord,
        get_pp pps ps = Some pp ->
        nth_error orders (part_order_idx p) = Some p_ord ->
        nth_error orders (part_order_idx pp) = Some pp_ord ->
        (* the valid field of the current participation is true *)
        ord_rt_valid p_ord = true ->
        (* tokens match *)
        order_tokenS (ord_rt_order p_ord) =
        order_tokenB (ord_rt_order pp_ord) ->
        (* orders in the rest of participations are valid *)
        _ring_orders_valid orders (pps ++ p :: nil) ps ->
        _ring_orders_valid orders pps (p :: ps)
  .

  (* Orders in ring 'r' are valid. *)
  Definition ring_orders_valid
             (r: RingRuntimeState)
             (orders: list OrderRuntimeState) : Prop :=
    (* r.valid == true *)
    ring_rt_valid r = true /\
    let ps := ring_rt_participations r in
    (* ... and r has at least 2 and at most 8 participations (ps) *)
    1 < length ps <= 8 /\
    (* ... and all orders referred by participations are valid *)
    _ring_orders_valid orders nil ps.
```

2.4 RingCanceller

`RingCanceller` provides a set of public functions for brokers to make their orders to become canceled at the specified time. Each function provides a different way to specify orders, such as by the order hash, the order owner, the trading tokens of that order, and the certain combinations of them. If an order-ring contains orders that have been canceled before a message call to `submitRings()`, the entire order-ring will not be filled by `submitRings()`.

- Modeled contract files
  - `contract/impl/RingCanceller.sol`
- Coq files
  - `Models/{RingCanceller, States, Events, Messages}.v`

2.4.1 Sub-state

The sub-state of `RingCanceller` is represented by `wst_ring_canceller_state` in `WorldState` and defined by `RingCancellerState` in Coq file `States.v`. Each field of the sub-state represents a storage variable of `RingCanceller`.

2.4.2 Specifications

Each public cancellation function in `RingCanceller` is modeled by a specification in Coq file `RingCanceller.v` with a similar name*, e.g.,* `cancelOrders()` by `cancelOrders_spec`.

`RingCanceller` primarily calls `TradeDelegate` to accomplish various cancellation operations, so the specification of each cancellation functions is primarily defined by the model of `TradeDelegate`.

Take the contract function `cancelAllOrdersForTradingPair()` as an example. `cancelAllOrdersForTradingPair()` calls `setTradingPairCutoffs()` in `TradeDelegate`. Its specification `cancelAllOrdersForTradingPair_spec` refers to the model of `TradeDelegate` (defined in Section 2.5) for the requirements, state changes and generated events of the message call to `setTradingPairCutoffs()`, and use them to describe the behavior of `cancelAllOrdersForTradingPair`.

2.5 TradeDelegate

Both `RingSubmitter` and `RingCanceller` call `TradeDelegate` to accomplish their jobs, including

- transferring ERC20 tokens,
- canceling orders,
- recording filled amounts of orders, and
- querying filled amounts and cancellation status of orders.

Besides the query function, all other functions above are restricted to be called only by **authorized users** and when `TradeDelegate` is **not suspended or killed**. Therefore, `TradeDelegate` also provides public interface to

- authorize/deauthorize users
- suspend/resume/kill itself.

The **suspend/kill** of `TradeDelegate` actually **pauses/stops** LPSC from trading and canceling any orders.

- Modeled contract files
    - `contract/impl/TradeDelegate.sol`

- Coq files
  - `Models/{TradeDelegate, States, Events, Messages}.v`

2.5.1 Sub-state

The sub-state of `TradeDelegate` is represented by `wst_trade_delegate_state` in `WorldState` and defined by `TradeDelegateState` in Coq file `States.v`. Each field of the sub-state represents a storage variable of `TradeDelegate` with the similar name. For example,

- `delegate_owner` models the storage variable `owner` that specifies the contract owner of `TradeDelegate`;
- `delegate_suspended` models the storage variable `suspended` that specifies if the contract is suspended;
- `delegate_authorizedAddresses`, which maps addresses to boolean values, specifies whether an address is authorized and models the semantics of the storage variable `authorizedAddresses`.

2.5.2 Specifications

Each public/external function in `RingCanceller` is modeled by a specification in Coq file `TradeDelegate.v` with a similar name, *e.g.,* `batchTransfer()` by `batchTransfer_spec`.

Specially, restrictions on callers are modeled in `fspec_require` of specifications, which must be satisfied in order to execute the function without revert.

- For functions requiring authorized callers, `fspec_require` in their specifications include a proposition that states `sender` (representing `msg.sender`) must be mapped by `delegate_authorizedAddresses` in the pre-state to `true`;
- For functions limiting callers to the contract owner, `fspec_require` in their specifications include a proposition that states `sender` (representing `msg.sender`) must be the same as `delegate_owner` in the pre-state.

Restrictions on the suspended status are also modeled in `fspec_require`.

- For functions only working in the non-suspended status, `fspec_require` in their specifications include a proposition that states `delegate_suspended` in the pre-state must be `false`;
- For functions only working in the suspended status, the opposite proposition is included in `fspec_require`.

2.6 FeeHolder

In LPSC, all fees, including the wallet fee, the miner fee, the burned fee and fees shared to orders, are first transferred to `FeeHolder` which also records who has how much fee in which token. Every fee recipient can later call `FeeHolder` to withdraw its fees, *i.e.,* transferring fees in token contracts.

- Modeled contract files
  - `contracts/impl/FeeHolder.sol`
- Coq files
  - `Models/{FeeHolder, States, Events, Messages}.v`

2.6.1 Sub-state

The sub-state of `FeeHolder` is represented by `wst_feeholder_state` in `WorldState` and defined by `FeeHolderState` in Coq file `States.v`. Each field of the sub-state represents a storage variable of `FeeHolder` with the similar name. For example,

- `feeholder_feeBalances` is defined by a total map in Coq from the token address and the fee recipient address to the amount of remaining fees, which models the storage variable `feeBalances`.

2.6.2 Specifications

Each public/external function in `FeeHolder` is modeled by a specification in Coq file `FeeHolder.v` with the similar name, *e.g.,* `withdrawBurned()` by `withdrawBurned_spec`.

In the specification `batchAddFeeBalances_spec` of `batchAddFeeBalances(byte32[] batch)`, which is called by `RingSubmitter` to record the fee balances:

- The byte array parameter is modeled by its semantics after deserialization, *i.e.,* a list of tuples (`token`, `recipient`, `amount`) (`FeeBalanceParam` in Coq file `Messages.v`). Each tuple means the `recipient` has an additional `amount` of fee in `token`.
- `fspec_trans` specifies that, for each tuple (`token`, `recipient`, `amount`) in the argument, the fee amount `amount` is added to the existing fee amount recorded in `feeholder_feeBalances` for token `token` and recipient `recipient`.
- `fspec_require` includes a proposition `is_authorized wst sender` which states `bachAddFeeBalances()` can be called only by authorized users.

In the specification `withdrawBurned_spec` of `withdrawBurned(address token, uint value)`, which is called by `BurnManager` to transfer an amount `value` of fee tokens from `FeeHolder` to `BurnManager` in the token contract of `token`:

- `fspec_require` specifies the requirement for the authorized caller in the same way as `batchAddFeeBalances_spec`.
- `fspec_require` also includes a proposition to state the remaining fee balance of the caller in `token` in the pre-state must be not less than the amount `value`.
- `fpsec_trans` specifies a successful execution of this function must present **both**
  - a `true` return value, and
  - a successful execution of the `transfer()` function of the fee token contract that transfers the specified amount of tokens from `FeeHolder` to the caller.
- `fspec_events` specifies the successful execution of this function must emit an event `TokenWithdrawn` that specifies an amount `value` of `token` is withdrawn from the fee balance of `FeeHolder`.

In the specification `withdrawToken_spec` of `withdrawToken(address token, uint value)`, which can be called by anyone to transfer an mount `value` of fee tokens from `FeeHolder` to the caller in the token contract of `token`:

- `fspec_require` does not include any restriction on the caller.
- `fspec_require` specifies the same requirement for sufficient remaining fee balance as `withdrawBurned_spec`.
- `fspec_trans` specifies the state changes and return value of a successful execution in the same way as `withdrawBurned_spec`.
- `fspec_events` specifies the successful execution of this function must emit an event `TokenWithdrawn` that specifies an amount `value` of `token` is withdrawn from the fee balance of the caller.

2.7 BurnManager

`BurnManager` provides a public interface which can be used by anyone to actually burn the burned fees which were held by `FeeHolder`.

Note: In the audited version, only fees in LRC token can be burned. The burn operation is accomplished by `burn()` in LRC token contract.

- Modeled contract files
  - `contracts/impl/BurnManager.sol`
- Coq files

- `Models/BurnManager, States, Events, Messages}.v`

2.7.1 Sub-state

The sub-state of `BurnManager` is represented by `wst_burn_manager_state` in `WorldState` and defined by `BurnManagerState` in Coq file `States.v`. Each field of the sub-state represents a storage variable of `BurnManager` with the similar name.

2.7.2 Specifications

The only public function `burn(address token)` transfers burned fees from `FeeHolder` and burns them by `burn()` of LRC token contract.

In its specification `burn_spec`:

- `spec_require` does not include any restriction on the caller, *i.e.,* this function can be called by anyone.
- `spec_require` includes a proposition that requires the argument `token` must be the LRC token.
- `spec_trans` states a successful execution of this function must present
  - a `true` return value,
  - a successful execution of `FeeHolder::withdrawBurned()` that transfers all burned fee tokens in LRC from `FeeHolder` to `BurnManager`,
  - a successful execution of `burn()` of the LRC token contract that burns all transferred burned fee tokens from `BurnManager`.

2.8 BurnRateTable

`BurnRateTable` manages the burn rates of tokens.

- Modeled contract files
  - `contracts/impl/BurnRateTable.sol`
- Coq files
  - `Models/{BurnRateTable, States, Events, Messages}.v`

In Loopring protocol v2, a certain percentage, *i.e.,* the burn rate, must be reduced from the ring-miner fees. The reduced fees, *i.e.,* the burned fees, are burned finally and not paid to ring-miners.

All tokens are divided into four tiers, while each tier has a different burn rate from others as shown below.

Tier	Burn Rate for P2P Order	Burn Rate for Non-P2P Order
1	0.5%	5.0%
2	2.0%	20.0%
3	3.0%	40.0%
4	6.0%	60.0%

- LRC is fixed at tier 1.
- WETH is fixed at tier 3.
- Other tokens stay at tier 4, unless they are upgraded via `upgradeTokenTier()`.
- Besides LRC, WETH and tokens at tier 1, users can call `upgradeTokenTier()` to upgrade a token by one tier. The caller must pay **0.5%** of the total LRC supply for each upgrade. The paid LRC tokens are **burned**.
- Besides LRC and WETH, if a token is not at tier 4, it can stay at its current tier for 63,113,904 seconds (730 days, or 2 years). If no further upgrade is performed for that token before the expiry, the token will be downgraded to tier 4 after the expiry.

2.8.1 Sub-state

The sub-state of `BurnRateTable` is represented by `wst_burn_rate_table_state` in `WorldState` and defined by `BurnRateTableState` in Coq file `States.v`.

- `burnratetable_tokens`, defined as a total map in Coq from token address to its tier and validation information, models the storage variable `tokens`.

2.8.2 Specifications

Each public/external function in `BurnRateTable` is modeled by a specification in Coq file `BurnRateTable.v` with a similar name.

In the specification `getBurnRate_spec` of `getBurnRate(address token)`, which can be called by anyone to get `token`'s P2P and non-P2P burn rates in one `uint32` value:

- `spec_trans` specifies `tokens`'s P2P and non-P2P burn rates, which are calculated from the tier information recorded in `burnratetable_tokens` (*i.e.,* `tokens` in `BurnRateTable`), are returned in the highest 16-bit and the lowest 16-bit of one 32-bit integer.

In the specification `getTokenTier_spec` of `getTokenTier(address token)`, which can be called by anyone to get `token`'s tier:

- `spec_trans` specifies `token`'s tier recorded in `burnratetable_tokens` (*i.e.,*

`tokens` in `BurnRateTable`) is returned.

In the specification `upgradeTokenTier_spec` of `upgradeTokenTier(address token)`, which can be called by anyone who has sufficient LRC tokens to upgrade non-LRC and non-WETH `token` by one tier:

- `spec_require` states
  - the argument `token` must not be the same as any of 0, the LRC token address and the WETH token address,
  - the current tier of `token` must not be 1,
  - the caller has enough LRC tokens (0.5% of the current total amount of LRC tokens) to burn.
- `spec_trans` specifies a successful execution must preset
  - a successful message call to `totalSupply()` of the LRC token contract, which gets the current total supply of LRC tokens,
  - a successful message call to `burnFrom()` of the LRC token contract, which burns 0.5% of the total supply of LRC tokens from the caller,
  - a return value `true`.
- `spec_events` specifies the events generated from a successful execution must contain an event `TokenTierUpgraded` that specifies the upgrade token is `token` and the post-upgrade tier of `token` is just one tier higher than before.

2.9 BrokerRegistry

`BrokerRegistry` provides a public interface for order owners to register the information of their brokers to LPSC. `RingSubmitter` also queries `BrokerRegistry` for those information when validating submitted orders.

- Modeled contract files
  - `contracts/impl/BrokerRegistry.sol`
- Coq files
  - `Models/{BrokerRegistry, States, Events, Messages}.v`

2.9.1 Sub-state

The sub-state of `BrokerRegistry` is represented by `wst_broker_registry_state` in `WorldState` and defined by `BrokerRegistryState` in Coq file `States.v`.

- `broker_registry_brokersMap`, defined as a map from the order owner address to a map from the broker address to the broker information `Broker`, approximately models the storage variables `brokersMap` and `positionMap` together.

2.9.2 Specifications

In the specification `registerBroker_spec` of `registerBroker(address broker, address interceptor)`, which is called by the order owner to register its `broker` and an optional `interceptor` (if not `0`) for that broker to LPSC:

- `fspec_require` states
  - the argument `broker` must be non-zero,
  - `broker` must have not been mapped in `broker_registry_brokersMap`, *i.e.,* it must have not been registered.
- `fspec_trans` specifies `broker_registry_brokersMap` in the post-state of a successful execution of this function must map the caller and the specified `broker` to a `Broker` that contains the order owner address, the broker address and the interceptor address.
- `fspec_events` specifies a successful execution of this function must emit an event `BrokerRegistered` that specifies the caller registers a `broker` whose interceptor, if not zero, is `interceptor`.

In the specification `unregisterBroker_spec` of `unregisterBroker(address addr)`, which is called by the order owner to unregister its broker `addr` from LPSC:

- `fspec_require` states
  - the argument `addr` must be non-zero,
  - `broker_registry_brokersMap` in the pre-state must have mapped the caller and the specified broker `addr`, *i.e.,* the broker `addr` must have been registered.
- `fpsec_trans` specifies `broker_registry_brokersMap` in the post-state of a successful execution of this function must not map the caller and the broker `addr`, *i.e.,* the broker is indeed unregistered.
- `fspec_events` specifies a successful execution of this function must emit an event `BrokerUnregistered` that specifies the caller unregisters its broker `addr`.

In the specification of `unregisterAllBrokers_spec` of `unregisterAllBrokers()`, which is called by the order owner to unregister all its registered brokers from LPSC:

- `fspec_trans` states `broker_registry_brokersMap` in the post-state does not map any broker for the caller.
- `fspec_events` specifies the successful execution must emit and event `AllBrokersUnregistered` that specifies all brokers of the caller have been unregistered.

In the specification `getBroker_spec` of `getBroker(address owner, address addr)`, which checks whether the order `owner`'s broker `addr` has been registered and, if registered, get its interceptor:

- `fspec_trans` simply looks up `broker_registry_brokersMap` to get the information of the specified broker.
    - If the broker is not registered, `RetBrokerInterceptor None` must be returned which represent the return of (false, 0) in the contract.
    - If the broker is registered, `RetBrokerInterceptor (broker_interceptor broker_info)` must be returned which represents the return of (true, interceptor) in the contract.

`getAllBrokers(address owner, uint start, uint count)`, which can get addresses and interceptors of multiple brokers of the caller, is not modeled and manually inspected.

2.10 OrderBook

`OrderBook` provides a public order book, where order owners and brokers can submit their orders on chain, and others can query for order details by order hashes.

- Modeled contract files
    - `contracts/impl/OrderBook.sol`
- Coq files
    - `Models/{OrderBook, States, Events, Messages}.v`

2.10.1 Sub-state

The sub-state of `OrderBook` is represented by `wst_order_book_state` in `WorldState` and defined by `OrderBookState` in Coq file `States.v`.

- `ob_orders`, defined as a total map from the order hash to `OrderElem` (defined in Coq file `States.v`) which contains all booked information, models the storage variable `orders`.

2.10.2 Specifications

In the specification `submitOrder_spec` of `submitOrder(bytes32[] dataArray)`, which is called by order owners and brokers to submit orders to the order book:

- The byte array argument is modeled by its semantics after deserialization.
- `fspec_require` specifies that

    - the caller must be the same as either the order owner or broker,
    - `ob_orders` in the pre-state must have not been mapped by `ob_orders` in the pre-state, *i.e.,* the order has not been submitted to the order book.
- `fspec_trans` specifies a successful execution of this function must present
    - `ob_orders` in the post-state is updated from that in the pre-state by mapping the submitter order by its hash, *i.e.,* the submitted order is indeed recorded in the order book;
    - the hash of the submitted order is returned.
- `fspec_events` specifies a successful execution of this function must emit an event `OrderSubmitted` that specifies the hash of the submitted order.

In the specification `getOrderData_spec` of `getOrderData(address hash)`, which can be called by anyone to query for the order information by the order hash,

- `fspec_trans` looks up `ob_orders` for the queried `hash`.
    - If it is mapped, the detailed order information `ord` in the map must be returned in the form `RetOrder (Some ord)`, which represents the case a non-empty byte array containing the order information is returned in the contract.
    - If it is not mapped, `RetOrder None` is returned, which represents the case an empty byte array is returned in the contract.

2.11 OrderRegistry

`OrderRegistry` allows order brokers to register hashes of their orders to LPSC. If the hash of an order has been registered by its broker, `RingSubmitter` will not need to verify the signature of the order submitted by the ring-miner.

- Modeled contract files
    - `contract/impl/OrderRegistry.sol`
- Coq files
    - `Models/{OrderRegistry, States, Events, Messages}.v`

2.11.1 Sub-state

The sub-state of `OrderRegistry` is represented by `wst_order_registry_state` in `WorldState` and defined by `OrderRegistryState` in Coq file `States.v`.

- `order_registry_hashMap`, defined as a total map from the broker address and the order hash to a boolean value that indicates whether the order is registered, models the storage variable `hashMap`.

2.11.2 Specifications

In the specification `isOrderHashRegistered_spec` of `isOrderHashRegistered(address owner, bytes32 hash)`, which checks whether the order `hash` of broker `owner` has been registered:

- `fspec_trans` specifies the return value must be what `order_registry_hashMap` (*i.e.,* `hashMap` in the contract) maps for the broker `owner` and the order `hash`.

In the specification `registerOrderHash_spec` of `registerOrderHash(bytes32 orderHash)`, which is called by order brokers to register hashes of their orders:

- `fspec_trans` specifies `order_registry_hashMap` in the post-state is updated from that in the pre-state by mapping the caller (*i.e.,* the order broker) and the specified order `hash` to `true`.
- `fspec_events` specifies the successful execution of this function must emit an event `OrderRegistered` that specifies the order broker and the registered order hash.

3. Properties

One way to check whether the contract implementation does guarantee certain good behaviors is to prove its formal model can satisfy corresponding properties. The failure to prove a property usually implies bugs in the implementation, as long as the formal model is consistent with the implementation.

Properties proved during the audit are explained in the rest of this section.

3.1 Properties about Suspend and Kill

`TradeDelegate` provides `suspend()` and `kill()` to suspend and stop LPSC from filling and canceling orders. Following theorems proved in Coq file `Props/ControlProps.v` show the suspend and kill functions indeed work as expected.

- Theorem `no_further_LPSC_transaction_once_suspended`

  After `TradeDelegate::suspend()` is successfully called, if there is no further successful call to `TradeDelegate::resume()`, all further message calls to

  - `RingSubmitter::submitRings()`
  - all `cancel*()` functions in `RingCanceller`
  - `TradeDelegate::batchTransfer()`
  - `TradeDelegate::batchUpdateFilled()`
  - `TradeDelegate::setCancelled()` and all `set*Cutoffs*()` functions in `TradeDelegate`

  will always fail (revert), *i.e.,* no order can be filled or canceled.

- Theorem `no_further_LPSC_transaction_once_killed`

  After `TradeDelegate::kill()` is successfully called, regardless of whether `TradeDelegate::resume()` is called afterwards, all further message calls to

  - `RingSubmitter::submitRings()`
  - all `cancel*()` functions in `RingCanceller`
  - `TradeDelegate::batchTransfer()`
  - `TradeDelegate::batchUpdateFilled()`
  - `TradeDelegate::setCancelled()` and all `set*Cutoffs*()` functions in `TradeDelegate`

  will always fail (revert), *i.e.,* no order can be filled or canceled.

- Theorem `LPSC_cannot_be_resumed_once_killed`

After `TradeDelegate::kill()` is successfully called, all further calls to `TradeDelegate::resume()` will fail, *i.e.,* a killed `TradeDelegate` cannot be resumed any more.

3.2 Properties about Privileged Users

Some critical operations in LPSC must be operated by the contract owner or authorized users. Following theorems proved in Coq file `Props/ControlProps.v` show LPSC does guarantee them.

- Theorem `only_owner_is_able_to_control_LPSC`

  Only the contract owner who deploys `TradeDelegate` can suspend, resume and kill `TradeDelegate`. Combined with theorems in Section 3.1, we can conclude that only the contract owner can suspend, resume and stop LPSC from filling and canceling orders.

- Theorem `only_authorized_contracts_are_able_to_fill_or_cancel_orders`

  Functions in `TradeDelegate` that transfer tokens and cancels orders can only be called by authorized users.

3.3 Properties about Valid Order-rings

LPSC puts several restrictions on submitted order-rings that can be successfully filled. Following theorems proved in Coq files `Props/{RingSubmitter, FilledRing}Props.v` show LPSC does guarantee them.

- Theorem `no_sub_rings`

  If an order-ring, submitted via `RingSubmitter::submitRings()`, contains sub-rings, *i.e.,* a token is sold in more than one orders in that ring, it will not be filled by `submitRings()`.

- Theorem `no_cancelled_orders`

  If an order-ring, submitted via `RingSubmitter::submitRings()`, contains canceled orders, it will not be filled by `submitRings()`.

- Theorem `no_token_mismatch_orders`

  If an order-ring, submitted via `RingSubmitter::submitRings()`, contains adjacent orders of which the buying token of the first one is not the selling token of the second one, it will not be filled by `submitRings()`.

- Theorems `soundness` and `completeness` in `Props/FilledRingProps.v`

These two theorems prove that, in a simplified scenario where no fee and rounding error are considered, only the order-ring, of which the product of token exchange rates of all orders is not less than 1, can be finally filled by `RingSubmitter::submitRings()` and the filled token exchange rate of each order is not worse than the original one.

3.4 Properties about Order Cancellation

`RingCanceller` provides a set of functions for order brokers to cancel orders. Following theorems proved in Coq file `Props/RingCancellerProps.v` show all those functions cannot undo the cancellation of any canceled order.

- Theorem `cancelOrders_no_side_effect`

  Every order ever canceled by `cancelOrders()` remains canceled, and is not affected by subsequent calls to `cancelOrders()`.

- Theorem `cancelAllOrdersForTradingPair_no_side_effect`

  Every order ever canceled by `cancelAllOrdersForTradingPair()` remains canceled, and is not affected by subsequent calls to `cancelAllOrdersForTradingPair()`.

- Theorem `cancelAllOrders_no_side_effect`

  Every order ever canceled by `cancelAllOrders()` remains canceled, and is not affected by subsequent calls to `cancelAllOrders()`.

- Theorem `cancelAllOrdersOfOwner_no_side_effect`

  Every order ever canceled by `cancelAllOrdersOfOwner()` remains canceled, and is not affected by subsequent calls to `cancelAllOrdersOfOwner()`.

- Theorem `cancelAllOrdersForTradingPairOfOwner_no_side_effect`

  Every order ever canceled by `cancelAllOrdersForTradingPairOfOwner()` remains canceled, and is not affected by subsequent calls to `cancelAllOrdersForTradingPairOfOwner()`.

3.5 Properties about Fee Withdraw

`FeeHolder` provides `withdrawBurned()` and `withdrawToken()` to withdraw burned fees and fees. Following theorems proved in Coq file `Props/FeeHolderProps.v` show some of their properties.

- Theorem `withdrawBurned_noauth`

If the caller is not authorized, its call to `withdrawBurned()` can never succeed. That is, non-authorized users cannot directly withdraw burned fees from `FeeHolder`.

Note: It does not mean non-authorized users cannot burn the burned fees. They can still call `burn()` of `BurnManager` to burn the burned fees in LRC. `BurnManager` is authorized.

- Theorem `withdrawBurned_auth`

  If the caller is authorized and its call to `withdrawBurned(token, amount)` succeeds, then the following three effects must **all** present:

  - the reduction of the specified `amount` of `token` from the corresponding burned fee balance of `FeeHolder`,
  - a return value `true`, and
  - an event `TokenWithdrawn` that specifies an `amount` of `token` is withdrawn from the fee balance of `FeeHolder`.

- Theorem `withdrawToken_noauth`

  A successful call to `withdrawnToken(token, amount)` must present **all** following three effects:

  - the reduction of the specified `amount` of `token` from the corresponding fee balance of the caller,
  - a return value `true`, and
  - an event `TokenWithdrawn` that specifies the caller, the fee token, and the withdrawn amount.

3.6 Properties about Fee Burning

`BurnManager` provides a public function `burn()`, which can be called by anyone to burn LRC fees from `FeeHolder`.

- Theorem `BurnManager_decrease_balance_of_fee_holder`

  This theorem in Coq file `Props/BurnManagerProps.v` proves that `burn()` only decreases the ERC20 balance of `FeeHolder`. ERC20 balances and allowances of other users are not affected.

4. Issues

Following issues have been found in the audit. All of them have been fixed in the latest version of LPSC, or have been addressed out of LPSC, or have negligible effects in practice.

4.1 Issue #1 Mismatched tokens in adjacent orders

- **Description**: Each order specifies which token it wants to sell (denoted as `tokenS`), and which token it wants to buy (denoted as `tokenB`). Those two tokens are not necessarily the same. When `submitRings()` transfers the `tokenS` of the current order to the previous order, it must ensure `tokenB` of the previous order and `tokenS` of the current order are the same one. Otherwise, the wrong token may be transferred. However, such check is missed in the audited code.
- **Possible Solution**: Add the missed check in `RingSubmitter::submitRings()`
- **Status**: Fixed
- **Severity**: High
- **Misc**: This issue was found in the review of the specification of `submitRings()` (in Section 2.3) with the whitepaper. All sub-specifications abstract from `submitRings()`, especially those including the ring/order validation checks, do not put any restriction on the types of tokenS/tokenB in adjacent orders in the same ring, which is not consistent with examples in the whitepaper.

4.2 Issue #2 withdraw() does not handle external call failures properly

- **Description** `FeeHolder::withdraw()` is an internal function used to implement the public fee withdraw interface `withdrawBurned()` and `withdrawToken()` of `FeeHolder`. In the audited code, `withdraw()` works as below:

  1. It first reduces the withdrawn amount from the fee balances of `FeeHolder`.
  2. It then calls the external token contract to make the actual fee token transfer.
  3. If the external call succeeds, it will return `true`. Otherwise, `false` is returned.

  In step 3, when the external call fails, `withdraw()` forgets to undo the state changes ever made to the fee balances in step 1 before return. As a result, the caller of `withdrawBurned()` or `withdrawToken()` in such cases does not get that amount of fees and will never get, because `FeeHolder` thinks that amount of fees have been withdrawn according to its stale fee balances.

- **Possible Solution**: Revert on external call failures in `withdraw()`
- **Status**: Fixed
- **Severity**: High
- **Misc**: This issue was found when we attempted to prove theorem `withdrawToken_noauth` (in Section 3.5). For `withdrawToken()` in the audited code, its formal specification (specially the `fspec_trans` part) states a successful execution of `withdrawToken` may present the `false` return value with a `FeeHolderState` corresponding to the state after above step 1.

  However, the original version of theorem `withdrawToken_noauth` includes a conclusion that states a `false` return value must always come with a `FeeHolderState` which is same as the one before calling `withdrawToken()`. The proving based on the above specification got stuck quickly, because the above specification apparently could not offer what the conclusion expected.

4.3 Issue #3 Incomplete check for multiple all-or-none orders in multiple rings

- **Description**: If an all-or-none order cannot be fully filled, then LPSC should skip it. In the audited code, after calculating filled amounts and fees of all orders, `RingSubmitter::submitRings()` does iterate all submitted orders and checks whether every all-or-none order is fully filled. If an all-or-none order fails the check, the order and rings containing it will be skipped.

  However, such check misses the following case

  - One all-or-none order `Order0` is first checked, and can be fully filled by multiple order-rings.
  - Another order-ring containing `Order0` contains another all-or-none order `Order1`.
  - `Order1` is checked later and cannot be fully filled.

  Because the second order-ring will be skipped, `Order0` is actually not fully filled. The above check, which does not consider order-rings at all, is not able to capture such cross-ring influences. As a result, `submitRings()` will partially fill the all-or-none `Order0` at the end.
- **Possible Solution**: When checking all orders, if a non-fully filled all-or-none order is found, revert all filled amounts of orders in the same rings and redo the check for all orders.
- **Status**: Fixed
- **Severity**: Medium

- **Misc**: This issue was found when reviewing a simplified algorithm of `submitRings()` in which no fee was considered.

4.4 Issue #4 Potential GAS attack to ring-mines from order brokers/owners

- **Description**: Though ring-miners may check order owners' token balances to ensure produced order-rings can be filled, a malicious order owner/broker is still able to decrease its token balance before the ring-miner submits order-rings that contains its orders. At the end, those rings and possibly other rings as well will not be able to be filled, and the GAS paid by the ring-miner to call `submitRings()` are wasted.
- **Status**: Loopring developing team replied that the ring-miner in practice submits rings very soon after checking token balances, so the attack window is very short. Therefore, such GAS attack is rarely impossible in practice. SECBIT team thinks the explanation is reasonable.
- **Severity**: Low

4.5 Issue #5 High GAS consumption from abnormal token contracts

- **Description**: LPSC calls some external token contract functions (such as `transfer()`, `transferFrom()`, `allowance()` and `balanceOf()` of ERC20 token contracts) without setting GAS limitation. A malicious or hacked token contract may include high GAS operations. If such token is included in any submitted order, callers to some LPSC functions, such as ring-miners to `submitRings()`, `BurnManager` to `withdrawBurned()`, and brokers to `withdrawToken()`, may have to pay a large amount of GAS.
- **Status**: Loopring developing team replied the abnormal tokens are detected and filtered by relays.
- **Severity**: Low

4.6 Issue #6 Inaccurate filled amounts caused by rounding errors

- **Description**: Integer divisions in Solidity have round errors, which may return smaller results. Consider a ring of two orders as below is submitted via `submitRings()`:

  - `order0`: sell 1 token A, buy 10 token B, its spendable token A is larger than 1
  - `order1`: sell 10 token B, buy 1 token A, its spendable token B is just 5

  After the manipulation of `setMaxFillAmounts()`, the filled amounts of two orders become

  - `order0`: fillAmountS is 0, fillAmountB is 5
  - `order1`: fillAmountS is 5, fillAmountB is 0

Therefore, after the final token transfer, `order1` sells 5 token A, but it does not get any token B.

- **Status**: Loopring developing team replied the unit of numbers in the above case is actually in `wei`. The decimals of tokens in the real world are usually way large than 1 (usually ≥ 18, *i.e.,* 1 token $\geq 10^{18}$ wei), so the loss in practice is pretty small and can be ignored.
- **Severity**: Low

5. Best Practices

LPSC is really in a high quality and we think the following best practices need to be highlighted.

5.1 GAS Optimization

Structures of order and order-ring in LPSC are large, but multiple optimizations have been applied to reduce the GAS consumption.

- If a public function requires information of `Order` or `Ring`, its arguments only include the necessary fields and pack them in byte arrays. Such optimization can reduce the size of calldata and save the GAS consumption of users.
- Always allocate large structures in the cheap memory rather than the expensive storage, and use references as much as possible.
- Implement some GAS-consumed operations in assembly, in order to avoid the redundant and inefficient bytecode generated by the compiler.

5.2 Use of SafeMath

Integer overflow/underflow is a popular source of a lot of security issues. LPSC heavily uses SafeMath which is helpful to mitigate undiscovered integer overflow/underflow bugs.

5.3 Safe External Calls

LPSC needs to call external contracts, including broker interceptors and token contracts, which are not part of LPSC and trustless. For external calls performing critical operations, such as `transfer()` and `transferFrom()` of ERC20 token contracts, and `getAllowance()` and `onTokenSpent()` of broker interceptors, LPSC uses the low-level `call` to invoke them and check the return value of `call`, which are implemented in files `contracts/lib/ERC20SafeTransfer.sol` and `contracts/impl/BrokerInterceptorProxy.sol`. Therefore, LPSC can have a chance to handle the external call failures, rather than just revert everything.

In addition, the safe wrappers for broker interceptors set the GAS limit in order to avoid abnormal interceptors consume too much GAS.

5.4 Re-entry Check

`RingSubmitter::submitRings()` and its sub-calls may call external contracts. If those external contracts call `submitRings()` again, the integrity of `submitRings()` may be broken. `submitRings()` deploys a re-entry check by using the highest bit of a storage variable `ringIndex` in `RingSubmitter` as a flag to indicate if it is still running. This bit is checked at the beginning of `submitRings()`. If it is set which implies `submitRings()` is re-entered, then a revert will happen. If it is not set which means this is a fresh call, then the bit will be set. Before `submitRings()` returns, this bit is cleared.

5.5 Good Test

Loopring developing team also rewrote LPSC in TypeScript. Lots of tests have been performed on the TypeScript version, and used to check whether the Solidity version works as expected.

5.6 Comprehensive Whitepaper

The whitepaper of Loopring protocol thoroughly explained the Loopring ecosystem, the economic model and the lots of technical details, which can help a lot for auditors, users and investors to understand Loopring protocol.

6. Conclusion

Though 3 implementation errors and 3 potential risks were found in the audit, all of them have been fixed, or have been addressed out of LPSC, or have negligible effects in practice. In addition, LPSC employed lots of best practices in the development and has a good whitepaper.

In conclusion, LPSC is in a very good quality.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security and performability in code quality, logic design and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company or investment.

APPENDIX

Severity Levels

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock ethers inside the contract.
Medium	Damage contract's security under given conditions and damages the benefit of stakeholders.
Low	May cause damages in theory that, however, are impossible or can be ignored in practice.

SECBIT Lab is devoted to construct a common-consensus, reliable and ordered blockchain economic entity.

http://www.secbit.io

audit@secbit.io

@secbit_io