SN LIMIT ORDERS 4
23rd August, 2022

Hey Baedrik, I'm not sure what your process is for checking a smart contract, so I thought I'd attempt to make the job easier for you by explaining the reasons behind some of the code.

Why smart contract?
- Allows user to submit a limit order.
- User's funds can't be stolen like with a centralised exchange.

How it will be used
- User puts in a limit order
- Admin is able to get the details of orders.
- Only whitelisted addresses are able to fill orders. This is done so that user's will always pay a consistent gas fee.

With these general ideas in mind, I'll break down the functions one by one. I will skip going through generic functions handle, query and receive.

Functions:

 # fn init

- Sets the fields of the public config.
- Reasons for each field will become evident in other functions.

 # fn activity_records

- Checks that it can only be called by the admin. It does this by checking that the viewing key for BUTT for the admin is correct.
- It grabs the type of activity records requested.
- There are two types of activity records, stored in separate arrays. One is to record and order fill activities and the other is to record any order cancellations. It was designed this way so that if at any point the array for fill activities maxed out, it would still be possible to cancel orders so that user's could get their crypto back.

 # fn set_execution_fee_for_order

- Checks that only SSCRT is being sent in.
- Checks that the amount sent in matches the execution set in the config.
- Grabs the most recently created order for the caller.
- Raises an error if
  - the caller has not created any orders
  - if the fee is not set in the same block as when the order is created
  - if the order already has an execution fee associated with it
  - if the order is cancelled
  - If the order has already been filled to some amount
- It updates the order stored in the user's orders array and also the contract's orders array.
- It returns the user's humanized order so that it can be used in the UI straight away.

 # fn append_activity_record

- Stores an activity record into the appropriate array.

 # fn append_order

- Stores an order into the appropriate array.

 # fn calculate_fee

- Calculates the fee based on the user's BUTT balance and the to amount.
- This is used to calculate the net_to amount.

 # fn cancel_order

- Gets the order at the position in the user's orders array.
- Checks:
  - The order at the position exists for the caller.
  - the user isn't sending any amount in.
  - the token used matches the from token of the order that the user is cancelling.
  - The order isn't cancelled
  - The order isn't totally filled
- Reduces the recorded users' balance for the refunded token by the unfilled amount of the order.
- It refunds the unfilled amount back to the caller.
- It sets the order to cancelled and stores it in the user's orders array and the contract's orders array.
- It creates an activity record and stores it.
- If the order has an execution fee associated with it and it hasn't been spent, it refunds that too.

 # fn create_order

- It checks:
  - The to_token is registered.
- It calculates the fee for the order.
- It increases the recorded users' balance (sum_balance) for the token being sent in.
- It stores the order for both the contract and the creator.
- It sends the humanized creator's order back in the response to be used in the UI.

 # fn fill_order

- It checks:
  - caller's address is in the whitelist.
  - the amount being sent in is greater than zero.
  - The token sent in is the to token for the order.
  - The order hasn't been cancelled
  - The amount is less than or equal to the unfilled amount.
- It sends the execution fee associated with the order to the caller, if the order has not been filled at all yet. If this fill is part of a route sequence, it sends the fee to the caller of the route sequence.
- It increases the order's net_to_amount_filled by the amount sent in.
- It increases the order's from_amount based on the ratio of the net_to_amount_filled
- It updates the creator and contract orders.
- It sends the from token filled to the caller.
- It sends the to token filled to the creator.
- It reduces the from_token's sum balance by the from filled amount.
- It creates an activity record and stores it in the fill records array.

 # fn get_activity_records

- You'll be familiar with this from snip-20 txs

 # fn get_next_activity_record_position

- You'll be familiar with this from snip-20 txs

 # fn get_next_order_position

- You'll be familiar with this from snip-20 txs

 # fn get_orders

- You'll be familiar with this from snip-20 txs

 # fn order_at_position

- You'll be familiar with this from snip-20 txs

 # fn orders

- You'll be familiar with this from snip-20 txs.
- The only difference is that it uses the user's butt viewing key to access.

 # fn orders_by_positions

- Checks the user can access using the user's butt viewing key
- Grabs the orders for the user by the positions specified.

 # pad_response

 # query_balance_of_token

- Made this a separate function so that it can be used for writing tests.

 # register_tokens

- Checks that admin is calling.
- If the token hasn't been registered yet, it registers the token for the contract and also created a RegisteredToken.
- It sets the viewing key for the token for the contract.

 # rescue_tokens

- Checks that admin is calling
- If contract has any native tokens such as scrt, atom, it sends it to the admin.
- If called for a snip-20, it checks the balance of the contract for that token versus the sum_balance for that token (users' deposits balance) and sends the difference to the admin.

 # space_pad

 # update_config

- Checks that admin is calling.
- If updating the addresses allowed to fill
  - It auto adds the contract address and admin address.
- Updates execution fee if provided.

 # update_creator_order_and_associated_contract_order

- Convenience method to update the user's order in their array, and the corresponding order in the contract's array.

ROUTE FILL

This is a 3 part algorithm conceptually similar to routed DEX swaps. This allows a whitelisted address to flash loan an amount from the contract so that it can fill limit orders and also interact with DEX trade pairs.

 # fn handle_first_hop

- Checks
  - It's being called by address in whitelist.
  - Route has at least 2 hops
- Creates and stores a RouteState to be used in the next two functions. The one that may need explaining is the minimum_acceptable_amount. This is necessary when a swap with a DEX trade pair happens as the return amount is subject to slippage.
- It sends the borrowed amount and token to the trade smart contract with the appropriate msg.
- It puts the finalise_route function at the end of messages to make sure everything has executed appropriately.

 # fn handle_hop

- It checks
  - Route state exists
  - That the caller is trade_smart_contract from the previous current_hop
    - If there are more hops
      - It checks that the next hop's from token matches the token that has been sent in.
      - If the next hop is to the limit order contract
        - Check if amount is greater than unfilled amount, and if so, send in the unfilled amount.
      - Send the amount and token to the next trade smart contract,
    - If there are no more hops
      - Check
        - that token received is the same as borrow token
        - check that amount is equal to or greater than the borrow amount
          - if there is a minimum_acceptable_amount
            Check that amount is equal to or greater than the minimum acceptable amount
      - If there is any excess, send to the initiator. This will be the fee.
- It updates the route state's current_hop with the next hop, and the remaining_hops.

 # fn finalize_route

This is the last step of a sequence of trades initiated by handle_first_hop, based on the same algorithm in the the dex aggregator contract. It is intended to always make sure that the route was completed successfully.

 # swap_msg

- It generates the swap message for the next hop