ERC-20 TOKEN DUTCH AUCTION MARKET REPORT

Jarrod Li (z5169806)

The following report is not intended as a substitute for viewing the relevant contract code, associated comments and intrinsic documentation.

Data Model

The user accounts, sell side exchange ('SSE') and buy side exchange ('BSE') are all represented as maps. The SSE maps token addresses to a second map, sell order IDs to SellOrder structs. A second map between sell order IDs and token addresses is maintained as an index for order management purposes only. A user's balance (the amount of ETH they have deposited), debtAccrued (the amount of ETH tied to an inchoate obligation to buy) and tokens (the number of a particular token type they own) are stored in the user account map. Both the SSE and BSE are represented as doubly linked lists. The BSE maintains a global chain of buy orders. Whenever the matching mode is initiated, the BSE is traversed (from head to tail by following the next attribute on each SellOrder), ensuring orders are fulfilled temporally. That is, older buy orders (closer to the head) are fulfilled first, followed by all later orders. The OrderStatus attribute on the BuyOrder struct facilitates the determination of whether a bid is open. Only open bids are matched with a sell order. For blind bids, the BuyOrder will not reflect the price or amount of a token. These values are only populated after a blind bid is revealed.

Off-chain computations

The formulation of a blind bid requires off-chain computations. For the purposes of this project, these off chain computations reside in the testing files. The only computation required, e.g., L:243-260 of dutch_auction_house.js, is the hashing of the token address, price, amount of a token and a nonce. A nonce is included to further dissuade attackers from attempting to formulate a scheme by which a bid may be unblinded by brute force attack because, for example, market prices are usually roughly estimable and token addresses do not change. The Keccack-256 hashing algorithm is selected due to its relatively good collision resistance. In JS, the relevant util method that implements this algorithm is web3.utils.solidity$ha3.

Requirements

The following requirements enumerate those outlined in Part 1 of the specification.

1. Buyers and sellers can create accounts with ETH or ERC-20 tokens and ETH via the methods create_account and create_token_account.
2. ERC-20 tokens are held in the DutchAuctionHouse contract. Ownership is transferred publicly through create_token_account and deposit_token. The transfer is handled internally through handle_transfer_token.
3. Buyers and sellers can deposit and withdraw ETH via deposit_funds and withdraw_funds.

4. Sellers can reduce the price of an offer through reduce_price. A check is performed to ensure the price is greater than the existing price.
5. Sellers can withdraw a sell offer through withdraw_sell_order.
6. Buyers can make blinding bids via buy.
7. Buyers can open a blind bid via open_buy_order.
8. Buy offers cannot be re-blinded. Such logic does not exist in the contract.
9. Buyers can withdraw any offer publicly through withdraw_buy_order. The actual handling of order removals takes place privately in handle_remove_order.
10. Modes are enforced through the during modifier. The modifier polls the current block timestamp in poll_mode to determine whether the mode should be updated. The actual determination of whether an operation can be performed aligns with the enum Modes.
11. The matching process takes place in init_match_phase. The internal brokering occurs in match_buy_sell_exchange according to the algorithms described in the previous section.
12. Negative balances cannot occur. Only uint (unsigned integers are allowed). Solidity will panic with an "Arithmetic overflow" if negative balances are attempted. In any event, the debtAccrued attribute prevents the opening of a bid that is not serviceable by the current account balance (i.e., requiring further deposits to become serviceable).
13. Maps were selected for global state structures to reduce the amount of gas otherwise required: with O(1) lookups. To reduce the amount of time looking for matching bids, several SSE chains exist. That is, instead of traversing a global SSE chain, only the SSE chain for a *relevant* token is traversed, saving gas.

    The SSE chain further reduces the gas required for each transaction by inserting sell orders in their correct temporal and value order. This means that a cheaper order will be inserted earlier into the SSE chain than a more expensive order. During the matching mode, the algorithm seeks the first matching sell order. And because it is guaranteed to occur in the correct place, fewer lookups are needed to find a matching offer.

Running cost analysis

The following are the estimated gas costs using eth-gas-reporter[1] where AUD amounts are determined by conversion from gas to ETH to AUD according to Google Market Summary conversion. The following numbers are true and correct as of 05 April 2023.

[1] https://github.com/cgewecke/eth-gas-reporter

Method	Avg cost (gas)	Avg cost (AUD)	Calls (#)
buy	194715	0.55	5
create_account	50569	0.14	5
deposit_funds	53968	0.15	4
deposit_token	76179	0.22	6
init_match_phase	275384	0.78	1
open_buy_order	120571	0.34	3
sell	264141	0.75	3
withdraw_buy_order	76466	0.22	2
withdraw_funds	52106	0.15	7
withdraw_sell_order	74316	0.21	1
withdraw_token	87259	87259	5
Total	5320824	15.12	

Notice that the init_match_phase and sell methods are incredibly expensive relative to the cost of running other DutchAuctionHouse operations. These gas costs align with the number of buy and sell orders available. Tokens that are more “popular” will have a disproportionately large number of traversals on the SSE.

Recall that these costs reflect a lower amount of computational complexity given the algorithmic and state performance improvements outlined in the previous section.

Security considerations

The DutchAuctionHouse contract uses global locks to ensure mutually exclusive access to certain parts of the code during runtime. Where an operation that changes the internal state of the contract takes place, a require_mutex modifier is added. This prevents unsafe access to the global data structures that could allow an attacker to exploit re-entrancy vulnerabilities.

The use of stronger typing for data structures and their constitutive elements attempts to prevent programmer error. As mentioned, unsigned integer use prevents attackers from exploiting overflow attacks. Further, the use of single modifiers minimises errors in ensuring that a method invocation must meet certain conditions.

The contract also specifies explicit visibility levels of functions. This prevents internal methods from manipulation by external actors. It also provides users with a better understanding of which methods they should use to interact with the system. Moreover, no external libraries are used to prevent supply chain attacks. Note that the imported solidity file from OpenZepplin is *not* used within the DutchAuctionHouse.

Finally, the internal seek_order_id method observes an issue with calculating and providing order IDs. That is, only incrementing order IDs, and relying on a wrap-around once the maximum uint size is reached, is not safe. Such a scheme is susceptible to overriding

previous buy and sell orders by overriding their existence in the global state structures. Thus, seek_order_id assigns the closest *unused* ID to the transaction. This relies on the intuition that older IDs are likely to become released first, and so there is little computation effort required to seek the next ID. However, over-time this can lead to fragmentation and, as such, a defragmentation process should be formulated in the future.

Suitability

The Ethereum platform is suitable for this buy/sell side market. Consider the fact that other platforms like Binance[2] levy some 0.35% in transaction fees for small transactions. Sending sell orders and matching only costs around AUD$1.53 in gas. This means that any transaction of an amount greater than ~AUD$4 is cheaper on the DutchAuctionHouse contract. Such a contract also removes the cryptocurrency intermediaries such as Coinbase and Binance who are single points of failure and prone to liquidity and cyber risks.

The protection of bidders' privacy stands as another significant benefit of adopting the Ethereum platform. In many cases, a lack of transparency from traditional auction houses raises questions as to how user data is being handled. By providing certain mathematical guarantees, the Solidity contracts may very well prove acceptable to even the most privacy conscious individuals who may otherwise wish to refrain from broader participation. Furthermore, the transparency of an immutable ledger allows market participants to audit the market in operation. This provides yet another haven for sceptical participants. In turn, this also reduces the risk of fraudulent transactions and money laundering.

Testing

The methodology behind testing the DutchAuctionHouse contract largely treats the method logic as a black box. As such, asserts are used sparingly because they tend to check the internal state of the contract directly. Consider test 3 (as outlined below). It simply checks that the user cannot deposit funds without an account and that a user cannot withdraw their account balance. These tests do not check the state of the contract directly. They infer the state of the contract according to whether an operation succeeds or reverts. Thus, the use of expectRevert is used quite often throughout each of the following tests.

Test 1 (should setup the account state correctly)

This test is only used for setting up the initial contract state. It awaits the deployment of the DutchAuctionHouse contract. It awaits the creation of five accounts. It awaits the creation of a mock ERC20 token. It distributes the ERC20 tokens to the sellers.

Test 2 (should handle account creation correctly)

This test checks that a user can only open a single account.

[2] https://www.binance.us/fee/schedule

Test 3 (should handle ETH deposits and withdrawal correctly)

This test checks that a user can only deposit or withdraw ETH with an open account. It also verifies that a user cannot withdraw more funds than their account balance provides.

Test 4 (should handle token deposit and withdrawal correctly)

This test checks that the depositing and withdrawing of an ERC20 token is handled properly. The test deposits all tokens available to the sellers. Following this, the test withdraws all tokens available to the sellers on the exchange.

Test 5 (should handle place and withdraw sell orders correctly)

This test checks that a user can place and withdraw a sell order. It checks that a user cannot:

- sell a token they do not own;
- sell an amount of tokens exceeding those available to them; and
- withdraw a sell order that they did not create.

Test 6 (should handle place, withdraw and reveal buy orders correctly)

This test checks a user can place, withdraw and reveal a buy order. It checks that a user cannot:

- open a buy order by providing a different token type, price or amount;
- open a buy order of which they cannot service (in the sense that their account balance cannot fulfil the order); and
- withdraw a buy order that they did not create.

Test 7 (should handle matching buy and sell orders successfully)

This test checks that the execution of buy and sell orders occurs correctly. After executing the matching process it:

- attempts to withdraw all created orders that should have been fulfilled; and
- tries to withdraw funds that should have been transferred to a seller.

Stretch goal

To hide the true account owner while placing a blind buy order, a third-party account submits a blind order. This third-party account also provides a signature that is later used by ecrecover to recover the public key for the transaction, thereby determining the true buyer's address based on the private key used to sign the order.

As part of the stretch goal, the following test was added to the testing suite.

Test 8 (should handle third party blind-bids)

This test checks that a third party can place a blind bid on behalf of another person. It does this by placing a blind buy order as buyer 1, signed as the wealthy observer. Buyer 1 then attempts to open the signed buy order, but such an operation fails with “Signature recovery failure”. Following this, the wealthy observer opens their bid, succeeding in doing so. Buy 1 tries to withdraw the bid they made, however, this fails because the signature does not match their account. Finally, the wealthy observer withdraws this bid, also succeeding in doing so.