horizontal line

**BURSADEX questionary**

# Looking at the bursadex code, you didn’t use the safeMath for the overflow protection. Why?

SafeMath library is not used intentionally. Bursadex contract is supposed to be called by other contracts including the planned margin trading contract to lend tokens and to short tokens. So, the “require” approach used by safeMath library breaks the workflow. The cases of overflow can be handled using predefined return values.

# Usually, loops are not used in smart contracts, but they were used in the bursadex. Why did you use it? Can you get rid of it? What are the pros and cons of doing that?

Loops are used particularly for cycling through the order book. It really makes no problem when we are talking about constant methods. When it comes to methods that consume gas, the con of the cycles is obvious: the quantity of gas consumed becomes unpredictable. However, cycles are used in willbuy() and willsell() methods, specifically for the case that specified order slot is already filled. For this case, we try the next slot. Then, if full, the next one. And the next one… Here comes the cycle.

Another case of cycles use is buyAll() and sellAll() methods. In those methods, the contract runs through the list of order indexes, executing the orders unless the necessary amount is bought (or sold). In this case, gas amount used is quite predictable, when the list of order indexes is finite.

# Test is not included. Test is an important means of demonstrating the trust of smart contracts. Why is it not included?

I evaded using truffle at that time, the software wasn’t mature enough. I used manual testing instead. Another reason for that is that I sometimes white contracts in plain EVM code, for which I have an assembler. Truffle doesn’t support that.

# Does findBestAsk or findBestBid work properly? The array is not sorted and the user decides the order\_spot when make transaction of willsell or willbuy. Does that logic make sense? What if the order is filled in the middle of array?

The array cannot be sorted in the state because every change there costs a serious amount of gas. Technique of placing orders is: every order is placed on the closest free spot. To find the free spot, a constant method is used, willbuyFindSpot() and willsellFindSpot() accordingly. You should not place orders in the middle of the state, if you do this, there is a chance that the order will never be executed. The findSpot methods should be called. So, user does not decide the order\_spot, he calculates it with a constant method. It would cost a lot of gas to iterate the order book otherwise.

# Willsell, willbuy In both cases, should ether be transferred to fund? Why should it be?

You can place order to buy tokens even when you haven’t sent any ether to the contract. However, your order will be nullified if someone tries to sell you the tokens. There are three ways to send ether to the contract, you can just send it to the contract, you can send it with deposit() function or you can send it while placing an order. The brilliance of the scheme is that you really need just one call to place an order to buy tokens. It’s a feature, not a bug.

# Willsell or willbuy looks for making the maker order, but there are no emmited event. How can user make the taker order?

All orders are visible, the frontend should be monitoring the state. Orderbook is designed to be compact, you just iterate from the start unless you meet zero. Orderbook can grow if all spots are full, in this case the new spot is opened. If the frontend would ever need an event to update, it’s easy to add that. However, it costs gas to the users.

# When placing an order, do not lock the token. What problems can arise and who loses in that case? Is this reasonable?

You can place an order to sell your token, then transfer it somewhere else, but your order will stay. However, your order will not be found by constant methods (because they check if you really have the token). FindSpot methods will recommend your spot for new orders, so it can be overwritten at any time. If someone tries to execute your order, your order will be nullified. So, why lock the token? Besides, if user removes some of his tokens, the order is still valid for the rest of the tokens, which is pretty convenient. For example, I want to sell 100 dai but then I need to spend 5 dai somewhere. My order remains valid, for 95 dai, same price in ether.

# Is Using willbuyFindSpot recommend? Is it forced? Can you expect this to keep the system in the correct state?

Yes, it is expected to always use findSpot methods. They keep orderbook compact and manageable. However, you cannot break orderbook by not using them, it is just your orders that will be invisible outside of the bounds.

# One function is too long, and several functionalities are mixed. For example, if you look at the buy function, there is a cancel functionality. What are the advantages of coding like this?

It would remove the problem when someone tries to execute his own order. It’s not hard to add a method with similar functionality.

# Looking at the buy function,

# if (!Bursa(token).transferFrom(seller, buyer, amount)) return 0;

# Why token is cast to Bursa? Shouldn’t it be casting to ERC20?

ERC20 is not a token, it is a protocol. Bursa is a fully ERC20 compliant ether-pegged token used to represent ether balance of Bursa users.