Broxus TON-DEX contract analysis
Prepared by Pruvendo
2021/10/26

Executive Summary

In the course of this project, pursuant to the intention of the project authors, we have performed an audit of the Solidity smart contracts developed as a part of a decentralized token exchange (DEX). The document is intended for use by contracts' authors on their free will for public or private goals.

In the course of the audit, no major vulnerabilities with the Broxus smart contract ecosystem have been found. The few minor issues that do not imperil the Broxus smart contract ecosystem but still call for improvement have been listed in this document in the following sections.

Source data

The source code of the smart contracts is available on Github at:
https://github.com/broxus/ton-dex/tree/master/contracts
The commit hash 02943f5f05d280be795427f1adf01ab5f466616e made on 2021/08/11.
However, at the moment of submission of this document, two new commits have been made.

Motivation

The DEX ecosystem creation is one of the important goals of the Free TON network. Here we provide the public activity list organized as contests at DeFi sub-governance (https://defi.gov.freeton.org) :

1. #6 FreeTon DEX Architecture & Design
    a. Duration: October 28 - November 15, 2020, 23:59 UTC
    b. 10 positively scored submissions
    c. Winner (7.66 score): “Orderbook Dex Design and Architecture” at https://firebasestorage.googleapis.com/v0/b/ton-labs.appspot.com/o/documents%2Fapplication%2Fpdf%2Fin0jwkdweijkhjr6xdb-Orderbook%20Dex%20Design%20and%20Architecture.pdf?alt=media&token=84824133-7817-49ca-abd7-a4122ee8a900
2. #16 FreeTon DEX Implementation Stage 1 Contest
    a. Duration: January 27 - March 20, 2021, 23:59 UTC
    b. 7 positively scored submissions
    c. Winner (8.45 score): at https://firebasestorage.googleapis.com/v0/b/ton-labs.appspot.com/o/documents%2Fapplication%2Fpdf%2Fr5arcsbzehkm9nixl7-FreeDEX%20contest%20_%20SVOI%20submission%20_%20TonSwap.pdf?alt=media&token=4f1984d3-1c02-4c06-8c42-bd5d983af090
3. #23 FreeTon DEX Implementation Stage 2
    a. May 9 - June 18, 2021, 23:59 UTC
    b. 4 positively scored submissions
    c. Winner (8.12 score) at https://firebasestorage.googleapis.com/v0/b/ton-labs.appspot.com/o/documents%2Fapplication%2Fpdf%2Fjqyuwqjmvqgkq2ky5ej-II%20FreeDEX%20contest%20_%20SVOI%20submission%20_%20TonSwap.pdf?alt=media&token=d64e7847-24eb-4022-a722-99108c3751e5
4. #38 FreeTON DEX Implementation: Stage 3

a. August 4 - September 27, 2021, 23:59 UTC
b. 5 submissions
c. Voting in progress

The Broxus' contracts as a part of this activity and have the most mature system already running at production, please see information on the Web:

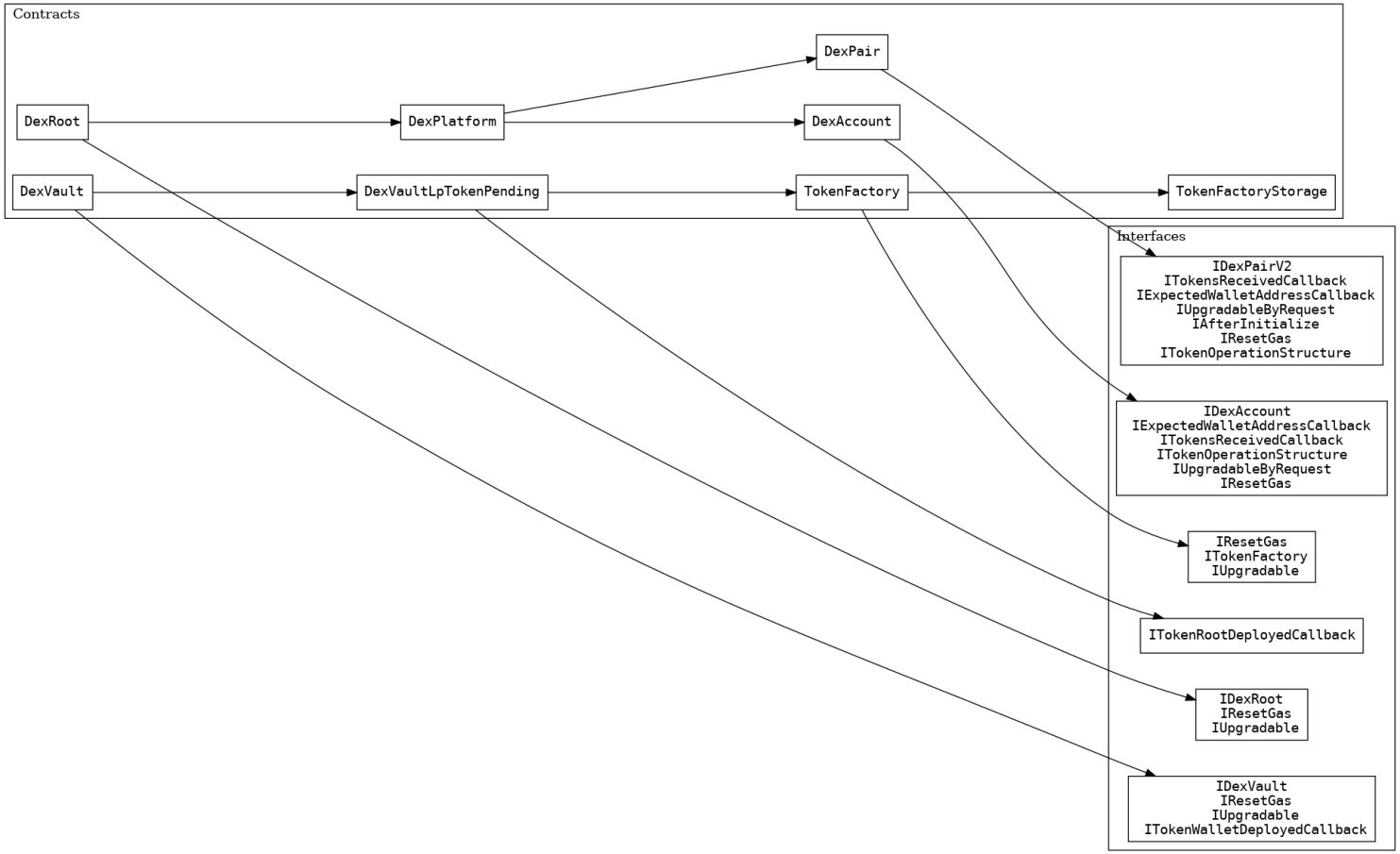
- https://broxus.medium.com/
- https://broxus.com/
- https://hub.forklog.com/broxus-zapustila-ton-swap-2-0-chto-novogo-v-dex-na-blokchejne-free-ton/ (in Russian)
- https://www.facebook.com/4BitcoinAddicts/posts/broxus-a-leading-developer-of-the-freeton-project-has-unveiled-the-second-versio/3064450753801402/
- https://twitter.com/broxus

As the main goal of exchanges is manipulating user tokens, the analysis of such contracts requires investigation to reduce the risk of malfunction and prevent possible attacks.

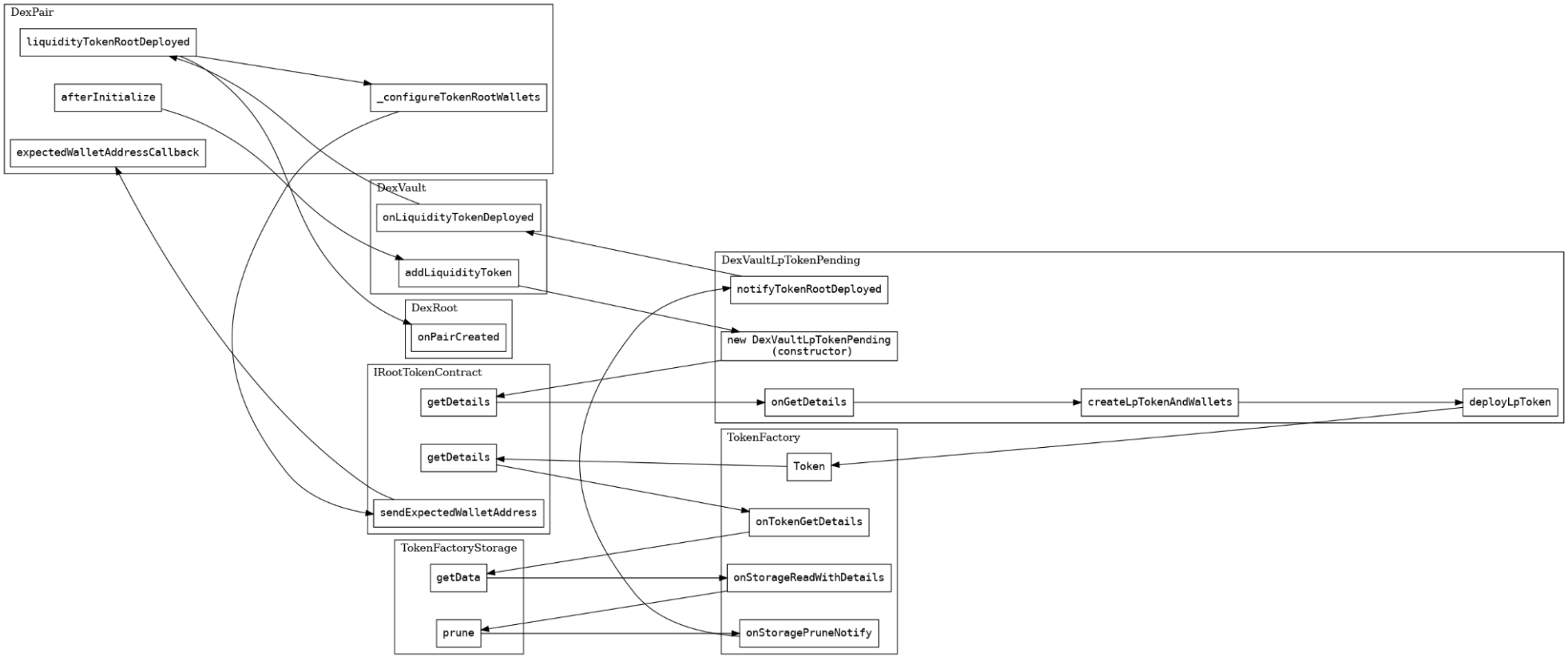
Program structure

This section contains the description of the contracts' hierarchy, the call graph (see the separate file) of one of the important call trajectories (liquidity provider token deployment).

Contracts' structure with interfaces and deploy sequence



LP token deploy trajectory DexPair -> …-> DexPair (with callbacks)



Issue List

Explanation

The contracts have been analyzed in terms of logical mistakes, whereas each function was analyzed in terms of readability, adequate variables' names, and functional encapsulation. Also, duplicate code was searched to prevent DRY[1] principle violation. In some cases, when a certain algorithm is implemented, common sense has been used to realize the correspondence of the final algorithm and potential goal. However, when common sense contradicts with the contract code, we cannot ensure that this is a bug, we mark that place to get more attention from the developers.

The majority of functions are well readable, variables' names are self-explanatory, the contract logic can be retrieved from its code with common knowledge of the Solidity programming language and the subject of the contract usage area.

We also follow the following principle: it's better to pay attention to a non-issue than to miss something important. Only the author of the contract can judge about the *real* severity of the issues found.

Common:

1. Since `responsible` is a relatively new feature, using it for getters is not commonly suggested as good practice, and some community members offer to use callbacks.
2. Using the flag `IGNORE_ERRORS` in message sending reduces strictness in terms contract behavior. If one suppose that errors can occur, we suggest to prevent them, using correspondent `requires`
3. Suggest to use bitwise operators like || for settings flags instead of arithmetic addition
4. Suggest not to use flag 2 in tvm.rawReserve, it reduces the contract behaviour determinism as it reserves the minimum from balance and given argument, and e.g. in the case when we reserve all the balance weCopy of Broxus DEX contract auditCopy of Broxus DEX contract auditë are out of funds to perform the function
5. `buildPairParams`, `buildInitParams`, `buildAccountParams` in every contract contain a lot of duplicated code (actually they are literally equal). We suggest to use a separate library or common ancestor
    a. `(tvm.hash(_buildInitData(`

[1] https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

```
        PlatformTypes.Account,
            _buildAccountParams(account_owner)))) -
```

suggest to perform the addresses calculation in separate function as it has duplicates

DexRoot:

6. deployAccount: During Account deployment the sender address is not taken into account, only the data passed as arguments. Suggest to make more checkups here.
7. deployAccount: improve checks for enough balance (only V1)
8. install*: suggest to check we have enough balance
9. The `forceUpgradeAccount` function is too rigid and lets root owner to perform an upgrade without letting the user to choose and leads to data lost
10. `setVaultOnce:` no check that `new_vault` is not zero
11. `requestUpgradeAccount:` doesn't emit events, however the similar functions (forceUpgradeAccount, upgradePair) do

DexAccount:

12. `getWalletData`: suggest to use optional type for return
13. `addPair`: The event emitted corresponds to the initial order of currencies (left and right roots) - not the order they are stored (which is sorted by address)
14. `expectedWalletAddressCallback` takes message value and not returns it back in a case of error (or send it to `send_gas_to`)
15. `tokensReceivedCallback:`

```
if(_balances.exists(token_root)) {
        _balances[token_root] += tokens_amount;
    } else {
        _balances[token_root] = tokens_amount;
    }
```

    a little redundant code , "+=" will work in both cases

16. `DexAccount.depositLiquidity:`

```
emit DepositLiquidity(left_root, left_amount, right_root, right_amount,
auto_change);
```

    `DexPair.depositLiquidity`:

```
emit DepositLiquidity(left_amount, right_amount, lp_tokens_amount);
```

    Same named events with different arguments

DexVault:

17. `constructor`: too public function which may potentially lead to many similar vaults

DexPair:

18. `exchange`: too duplicated code in branches, suggest refactoring
19. `setFeeParams (onlyRoot)`: root actually doesn't call this function

   a. suggest also checks: denominator > 0, numerator <= denominator (?)
20. afterInitialize: if the DexPair is not fully deployed in the case of Lp token unsuccessful deployment the only way to make it working is to call deployPair again and again, which is counter-intuitive as the Pair itself is actually deployed (but the Lp token).
   a. If something will happen with Pair (corruption of unknown reason) one can redeploy the same working Pair by the upgrade call, and after that call deployPair to initialize Lp root if it is zero (that is also a little confusing, however it is not an error)
21. tokensReceivedCallback: uses small TON values 10, 30, 44 (nanoton) to identify behaviour. Suggest to use a more self-explanatory way.
22. depositLiquidity:

```
require(lp_supply != 0 || (left_amount > 0 && right_amount > 0), DexErrors.WRONG_LIQUIDITY);
require((left_amount > 0 && right_amount > 0) || (auto_change && (left_amount + right_amount > 0)), DexErrors.AMOUNT_TOO_LOW);
```

can be refactored to

```
require((left_amount > 0 && right_amount > 0) || ((lp_supply != 0)&&(auto_change && (left_amount + right_amount > 0)), Error);
```

However it reduces error verbosity

23. depositLiquidity: The requirement of lp_supply != 0 seems technically reasonable but it is suggesting to use more economically inspired values as a lower bound (actually we do not see any good solution here so we can address this in the future of contract evolution)

DexVaultLpTokenPending:

24. constructor: suggest to use strict assignment of pending_messages = 2 rather than += to give more guarantees to the value assignment
   a. deploy_value is unused
25. Variables for LP symbol creation may be constants rather than variables:

```
string LP_TOKEN_SYMBOL_PREFIX = "TONSWAP-LP-";
string LP_TOKEN_SYMBOL_SEPARATOR = "-";
uint8 LP_TOKEN_DECIMALS = 9;
```

TokenFactory:

26. `onStorageReadWithDetails`: Usage of `TokenFactoryStorage(msg.sender).prune` callbacks (onStoragePruneNotify,

onStoragePruneReturn) seem redundant, getData callback (onStorageReadWithDetails) contains all the necessary parameters to determine the correct branch in function, and requires are the same. Suggest refactoring

Quadratic root and equation solving

We start with the quadratic root.
The method employed is the Babylonian method for finding square root adapted for integer-valued arguments. The algorithm is as follows:

```
function _sqrt(uint256 x) private pure returns (uint256) {
       if (x == 0) return 0;
       else if (x <= 3) return 1;
       uint256 z = (x + 1) / 2;
       uint256 y = x;
       while (z < y)
       {
           y = z;
           z = (x / z + z) / 2;
       }
       return y;
   }
```

That is commonly referred to as Newton's method (and also Heron's method as well). It is known that the convergence of Newton-like methods highly depends on the initial approximation. The less the difference between real square root and initial value is, the faster the method converges. In the contract, the initial value of (x + 1) / 2 is used. However, it is possible to find a more close to the real root alternative.

To motivate results let us use 2^[log2 X/2] as the starting value.

x = 12345678	**23,5575027=log2 -> (23+1)/2->12->2^12 = 4096**
1. 6172839	4096
2. 3086420	3555
3. 1543211	3513
4. 771609	3513
5. 385812	3513
6. 192921	3513
7. 96492	3513
8. 48309	3513
9. 24282	3513
10. 12395	3513
11. 6695	3513
12. 4269	3513
13. 3580	3513
14. 3514	3513
15. 3513	3513
16. 3513	3513

We see that, if we use the half of the argument (first column), convergence is achieved at the 16th step, while if we start from the 2^(([log2 (X)]+1)/2), the same is achieved at the 4th. [x] here means the floor function returning the highest integer, less than the argument. The value of [log2 (X)]+1 is just the number of binary digits in the number. There is an instruction in TVM

- B602 — BITSIZE $(x - c)$, computes smallest $c \geq 0$ such that x fits into a c-bit signed integer $(-2^{c-1} \leq c < 2^{c-1})$.

which returns this value. However, we know that, at some moment, there is no standard library function which does the same. It is also possible to find that value by the dichotomy procedure using right bit shift with O(log(log(X)) complexity.

To motivate more we take a bigger number to demonstrate convergence.

x = 12 345 678 912 345 600	**log2= 53,45485569**
1. 6 172 839 456 172 800	134217728
2. 3 086 419 728 086 400	113100101
3. 1 543 209 864 043 200	111128599
4. 771 604 932 021 604	111111112
5. 385 802 466 010 810	111111110
6. 192 901 233 005 421	111111110
7. 96 450 616 502 742	
8. 48 225 308 251 434	
9. 24 112 654 125 844	
10. 12 056 327 063 177	
11. 6 028 163 532 100	
12. 3 014 081 767 073	
13. 1 507 040 885 584	
14. 753 520 446 887	
15. 376 760 231 635	
16. 188 380 132 201	
17. 94190098868	
18. 47095114969	
19. 23547688556	
20. 11774106420	
21. 5887577482	
22. 2944837192	
23. 1474514752	
24. 741443729	
25. 379047296	
26. 205808791	
27. 132897475	
28. 112896869	
29. 111125233	
30. 111111111	
31. 111111110	
32. 111111110	

In the first case, the algorithm converges at the 32nd step while the log2-based variant on the 6th. So we suggest improving the initial approximation to make the convergence significantly faster.

Square equation solving

```
    function _solveQuadraticEquationPQ(uint256 p, uint256 q) private pure
returns (uint128) {
        uint256 D = math.muldiv(p, p, 4) + q;
        uint256 Dsqrt = _sqrt(D);
        if (Dsqrt > (p/2)) {
            return uint128(Dsqrt - (p/2));
        } else {
            return uint128((p/2) - Dsqrt);
        }
    }
```

1. uint256 D = math.muldiv(p, p, 4) + q; as (p:uint256) p^4+q can overflow uint256 size
2. As (D:uint256) sqrt(D) must fit uint128 size
3. if sqrt is working correctly Dsqrt > (p/2) always so the second branch shouldn't exist
    a. If it exists, it hints at an incorrectly working sqrt, and there are no reasons to assign anything but the 0 to the return value.
4. In any case, the value of uint128((p/2) - Dsqrt) is not a meaningful value (the second root is -(p/2) + Dsqrt, which is negative)
5. _sqrt function can return uint128 value instead of uint256

Conclusion

Thus far, we have found no major vulnerabilities that would imperil a client's funds involved while operating with Broxus. The system appears robust, applicable for usage in the Free TON ecosystem, and based on widely recognized Uniswap AMM.

Pros:

1. The contracts are well structured
2. Variables' names are self-explanatory in most cases
3. The events are sent, and they are satisfactory
4. All the functions carefully check the balance before transfers and reserve the minimum contract balance to prevent underflow
5. All the functions correctly check access rights
6. Almost all constants are named, and they are self-explanatory (there is a special library with constants designed)
7. Getters are complete to fulfil contract interaction
8. We suggest there is no need to redesign architecture
9. We have not found any issue which we can address as a critical or a major one

Cons:

1. Some functions are long enough (e.g. tokensReceivedCallback 437 LOC and some are 200+) and difficult to implement during code review
2. The intercommunication of contracts with callbacks sometimes takes long enough length (see Lp token deploy trajectory with 19 functions in a row)
3. Duplicate code occurs
4. Some issues are found and described at the preceding pages

All the issues found were discussed with the authors during Zoom calls and by Telegram channel.

Finally, we propose that a correspondent workaround is desirable and we can recommend the contracts to be used in production with the following proviso:

Some contract behaviour (maybe not full) is strongly suggested to pass formal verification, especially:

- functions with mathematical calculations;
- long contracts (more than 6 in a row) calls chain (including callbacks);
- deployment procedure and code upgrade effects.