Balancer Finance Process Quality Review

Score 74%

This is a Balancer Finance Process Quality Audit completed on July 2020. It was performed using the Process Audit process (version 0.3) then updated to V0.4 as the scroing weights changed. It is documented here. The audit was performed by ShinkaRex of Caliburn Consulting. Check out our Telegram.

The final score of the audit is 74%, an clear pass. The breakdown of the scoring is in Scoring Appendix.

Disclaimer

This report is for informational purposes only and does not constitute investment advice of any kind, nor does it constitute an offer to provide investment advisory or other services. Nothing in this report shall be considered a solicitation or offer to buy or sell any security, future, option or other financial instrument or to offer or provide any investment advice or service to any person in any jurisdiction. Nothing contained in this report constitutes investment advice or offers any opinion with respect to the suitability of any security, and the views expressed in this report should not be taken as advice to buy, sell or hold any security. The information in this report should not be relied upon for the purpose of investing. In preparing the information contained in this report, we have not taken into account the investment needs, objectives and financial circumstances of any particular investor. This information has no regard to the specific investment objectives, financial situation and particular needs of any specific recipient of this information and investments discussed may not be suitable for all investors.

Any views expressed in this report by us were prepared based upon the information available to us at the time such views were written. Changed or additional information could cause such views to change. All information is subject to possible correction. Information may quickly become unreliable for various reasons, including changes in market conditions or economic circumstances.

Executing Code Verified

This section looks at the code deployed on the Mainnet that gets audited and its corresponding software repository. The document explaining these questions is here. This audit will answer the questions;

1. Is the deployed code address(s) readily available? (Y/N)
2. Is the code actively being used? (%)
3. Are the Contract(s) Verified/Verifiable? (Y/N)
4. Does the code match a tagged version in the code hosting platform? (%)
5. Is the software repository healthy? (%)

Is the executing code address(s) readily available? (Y/N)

Answer: No

In fact, it was tricky to find the deployed address. There was no Developer page on the website. On the bottom there was a GitHub icon. From that, one of the pinned repositories was "balancer-core". Its readme had the docs.balancer.finance Gitbook. There is no contract address section in the GitBook. The address was indicated in the Bug Bounty section to give clarity to those hunting for a Bounty. This was not simple at all.

They are available at Address 0x9424B1412450D0f8Fc2255FAf6046b98213B76Bd as indicated in the Appendix. This Audit only covers the contract BPool. It was deployed on Feb 27, 2020.

How to improve this score

Make the ethereum addresses of the smart contract utilized by your application available on either your website or your github (in the README for instance). Ensure the address is up to date.

Is the code actively being used? (%)

Answer: 100%

Activity is well in excess of 100 transactions a day, as indicated in the Appendix. The ERC20 transactions exceed 100 a day but conventional transaction only take place a few times a day indicating a 2nd layer in use.

Percentage Score Guidance

100% More than 10 transactions a day

70% More than 10 transactions a week

40% More than 10 transactions a month

10% Less than 10 transactions a month

0% No activity

Are the Contract(s) Verified/Verifiable? (Y/N)

> Answer: Yes

0x9424B1412450D0f8Fc2255FAf6046b98213B76Bd is the Etherscan verified contract address.

Does the code match a tagged version on a code hosting platform? (%)

> Answer: 100%

Code verification was as smooth as possible. There was a complete release in the repository, but even without that the code matched the repsitory exactly.

GitHub address : https://github.com/balancer-labs/balancer-core/tree/master/contracts

Deployed contracts in the following file;

Balancer_Core.zip Balancer_Core.zip - 12KB

Release link: https://github.com/balancer-labs/balancer-core/releases/tag/v1.0.0

Is development software repository healthy? (%)

> Answer: 100%

balancer-core has over 900 commits on 3 branches.

Documentation

This section looks at the software documentation. The document explaining these questions is here.

Required questions are;

1. Is there a whitepaper? (Y/N)
2. Are the basic application requirements documented? (Y/N)
3. Do the requirements fully (100%) cover the deployed contracts? (%)
4. Are there sufficiently detailed comments for all functions within the deployed contract code (%)
5. Is it possible to trace software requirements to the implementation in code (%)

Is there a whitepaper? (Y/N)

> Answer: Yes

Location: https://balancer.finance/whitepaper/

Are the basic application requirements documented? (Y/N)

> Answer: YES

Location: https://docs.balancer.finance/

How to improve this score

Write the document based on the deployed code. For guidance, refer to the SecurEth System Description Document.

Do the requirements fully (100%) cover the deployed contracts? (%)

Answer: 55%

While the concept and math are well documented, there is little connection to the code. With the white paper and the docs, the basic math premise is very clearly described. In the smart contracts section they describe bout 40% of the functions in BPool.sol, However, even here there is too little connection of the code.

The code should be the implementation of the requirements. This means the requirements should fully cover the code such that the code is written from the requirements. This documentation is closer to explanation of the application. It is a very good description, though.

How to improve this score

This score can improve by adding content to the requirements document such that it comprehensively covers the requirements. For guidance, refer to the SecurEth System Description Document . Using tools that aid traceability detection will help.

Are there sufficiently detailed comments for all functions within the deployed contract code (%)

Answer: 45%

The code is quite well written. With clear variable names and straightforward style, a good bit of the code can be read. However many primary functions have virtually no code as indicated in

the Appendix. The exception are the math functions. All of them, especially BMath are very well documented.

Code examples are in the Appendix. As per the SLOC, there is 18% commenting to code.

How to improve this score

This score can improve by adding comments to the deployed code such that it comprehensively covers the code. For guidance, refer to the SecurEth Software Requirements.

Is it possible to trace software requirements to the implementation in code (%)

> ⚠ Answer: 0%

As indicated above, the docs are not well connected to the code. For this reason there is nothing close to traceability. In their defence, traceability is still a new concept for most of crrypto.

How to improve this score

This score can improve by adding traceability from requirements to code such that it is clear where each requirement is coded. For reference, check the SecurEth guidelines on traceability.

Testing

This section looks at the software testing available. It is explained in this document. This section answers the following questions;

1. Full test suite (Covers all the deployed code) (%)
2. Code coverage (Covers all the deployed lines of code, or explains misses) (%)
3. Scripts and instructions to run the tests (Y/N)
4. Packaged with the deployed code (Y/N)
5. Report of the results (%)
6. Formal Verification test done (%)
7. Stress Testing environment (%)

Is there a Full test suite? (%)

> Answer: 100%

The Balancer code suite is small; just 7 files and 1,035 lines. It is impressive what this small code set accomplishes. The test suite is larger but also compact. There is a 160% test line to code line ratio.

Code coverage (Covers all the deployed lines of code, or explains misses) (%)

> Answer: 96%

As per the GitHub Coveralls badge, there is a 96% code coverage ratio.

Scripts and instructions to run the tests (Y/N)

> Answer: Yes

The scripts to run the code are indicated in the readme.

Packaged with the deployed code (Y/N)

> Answer: Yes

Report of the results (%)

> Answer: 80%

The coveralls report indicates which lines are not covered. Given that the scripts to run the tests are included as indicated above, the report (while not in the form of a single report) covers all aspects except explanations for the misses (the code not covered).

Formal Verification test done (%)

Answer: No

No evidence of Formal Verification exists, but this is still a rare form of test.

Stress Testing environment (%)

Answer: 100%

A Kovan test network exists and is explained in the "Testing on Kovan" section of the docs.

Audits

Answer: 90%

Balancer had a top quality audit done just before deployment (JAN 2020) by Trail of Bits. They implemented many of the recommendations or had an explanation in their docs. The TOB audit left significant artifacts in the GitHub also.

In April 2020 a second audit was performed by Consensys Dilligence. The recommendations of this audit are not yet implemented but it indicates a continued improvement ethic by the developers of Balancer.

1. Multiple Audits performed before deployment and results public and implemented or not required (100%)
2. Single audit performed before deployment and results public and implemented or not required (90%)
3. Audit(s) performed after deployment and no changes required. Audit report is public. (70%)
4. No audit performed (20%)
5. Audit Performed after deployment, existence is public, report is not public and no improvements deployed (0%)

Appendices

Author Details

The author of this audit is Rex of Caliburn Consulting.

Email : rex@caliburnc.com Twitter : @ShinkaRex

I started with Ethereum just before the DAO and that was a wonderful education. It showed the importance of code quality. The second Parity hack also showed the importance of good process. Here my aviation background offers some value. Aerospace knows how to make reliable code using quality processes.

I was coaxed to go to EthDenver 2017 and there I started SecuEth.org with Bryant and Roman. We created guidelines on good processes for blockchain code development. We got EthFoundation funding to assist in their development.

Process Quality Audits are an extension of the SecurEth guidelines that will further increase the quality processes in Solidity and Vyper development.

Career wise I am a business development for an avionics supplier.

Scoring Appendix

Deployed Code Appendix

Code Used Appendix

Example Code Appendix

Good Commenting in BMath.sol

```
/**********************************************************************
// calcSingleOutGivenPoolIn
// tAo = tokenAmountOut            /      /
// bO = tokenBalanceOut           /      // pS - (pAi * (1 - eF)) \
// pAi = poolAmountIn            | bO - || ---------------------- | ^
// ps = poolSupply                \      \\          pS           /
// wI = tokenWeightIn      tAo =   \      \
// tW = totalWeight                    /     /      wO \       \
// sF = swapFee                 *  | 1 - |  1 - ---- | * sF  |
// eF = exitFee                        \     \      tW /       /
**********************************************************************/
function calcSingleOutGivenPoolIn(
    uint tokenBalanceOut,
    uint tokenWeightOut,
    uint poolSupply,
    uint totalWeight,
    uint poolAmountIn,
    uint swapFee
)
    public pure
    returns (uint tokenAmountOut)
{
    uint normalizedWeight = bdiv(tokenWeightOut, totalWeight);
    // charge exit fee on the pool token side
    // pAiAfterExitFee = pAi*(1-exitFee)
    uint poolAmountInAfterExitFee = bmul(poolAmountIn, bsub(BONE, EXIT_
    uint newPoolSupply = bsub(poolSupply, poolAmountInAfterExitFee);
    uint poolRatio = bdiv(newPoolSupply, poolSupply);

    // newBalTo = poolRatio^(1/weightTo) * balTo;
    uint tokenOutRatio = bpow(poolRatio, bdiv(BONE, normalizedWeight));
    uint newTokenBalanceOut = bmul(tokenOutRatio, tokenBalanceOut);

    uint tokenAmountOutBeforeSwapFee = bsub(tokenBalanceOut, newTokenBa

    // charge swap fee on the output token side
    //uint tAo = tAoBeforeSwapFee * (1 - (1-weightTo) * swapFee)
    uint zaz = bmul(bsub(BONE, normalizedWeight), swapFee);
    tokenAmountOut = bmul(tokenAmountOutBeforeSwapFee, bsub(BONE, zaz))
    return tokenAmountOut;
}
```

Little Commenting in BPool.sol

```
1     function joinswapPoolAmountOut(address tokenIn, uint poolAmountOut, uint
2         external
3         _logs_
4         _lock_
5         returns (uint tokenAmountIn)
6     {
7         require(_finalized, "ERR_NOT_FINALIZED");
8         require(_records[tokenIn].bound, "ERR_NOT_BOUND");
9
10        Record storage inRecord = _records[tokenIn];
11
12        tokenAmountIn = calcSingleInGivenPoolOut(
13                            inRecord.balance,
14                            inRecord.denorm,
15                            _totalSupply,
16                            _totalWeight,
17                            poolAmountOut,
18                            _swapFee
19                        );
20
21        require(tokenAmountIn != 0, "ERR_MATH_APPROX");
22        require(tokenAmountIn <= maxAmountIn, "ERR_LIMIT_IN");
23
24        require(tokenAmountIn <= bmul(_records[tokenIn].balance, MAX_IN_RATI
25
26        inRecord.balance = badd(inRecord.balance, tokenAmountIn);
27
28        emit LOG_JOIN(msg.sender, tokenIn, tokenAmountIn);
29
30        _mintPoolShare(poolAmountOut);
31        _pushPoolShare(msg.sender, poolAmountOut);
32        _pullUnderlying(tokenIn, msg.sender, tokenAmountIn);
33
34        return tokenAmountIn;
35    }
36
37    function exitswapPoolAmountIn(address tokenOut, uint poolAmountIn, uint
38        external
39        _logs_
40        _lock_
41        returns (uint tokenAmountOut)
42    {
43        require(_finalized, "ERR_NOT_FINALIZED");
44        require(_records[tokenOut].bound, "ERR_NOT_BOUND");
45
46        Record storage outRecord = _records[tokenOut];
47
48        tokenAmountOut = calcSingleOutGivenPoolIn(
```

```
                            outRecord.balance,
                            outRecord.denorm,
                            _totalSupply,
                            _totalWeight,
                            poolAmountIn,
                            _swapFee
                        );

        require(tokenAmountOut >= minAmountOut, "ERR_LIMIT_OUT");

        require(tokenAmountOut <= bmul(_records[tokenOut].balance, MAX_OUT_I

        outRecord.balance = bsub(outRecord.balance, tokenAmountOut);

        uint exitFee = bmul(poolAmountIn, EXIT_FEE);

        emit LOG_EXIT(msg.sender, tokenOut, tokenAmountOut);

        _pullPoolShare(msg.sender, poolAmountIn);
        _burnPoolShare(bsub(poolAmountIn, exitFee));
        _pushPoolShare(_factory, exitFee);
        _pushUnderlying(tokenOut, msg.sender, tokenAmountOut);

        return tokenAmountOut;
    }

    function exitswapExternAmountOut(address tokenOut, uint tokenAmountOut,
        external
        _logs_
        _lock_
        returns (uint poolAmountIn)
    {
        require(_finalized, "ERR_NOT_FINALIZED");
        require(_records[tokenOut].bound, "ERR_NOT_BOUND");
        require(tokenAmountOut <= bmul(_records[tokenOut].balance, MAX_OUT_I

        Record storage outRecord = _records[tokenOut];

        poolAmountIn = calcPoolInGivenSingleOut(
                            outRecord.balance,
                            outRecord.denorm,
                            _totalSupply,
                            _totalWeight,
                            tokenAmountOut,
                            _swapFee
                        );

        require(poolAmountIn != 0, "ERR_MATH_APPROX");
        require(poolAmountIn <= maxPoolAmountIn, "ERR_LIMIT_IN");

        outRecord.balance = bsub(outRecord.balance, tokenAmountOut);

        uint exitFee = bmul(poolAmountIn, EXIT_FEE);

        emit LOG_EXIT(msg.sender, tokenOut, tokenAmountOut);
```

```
104
105         _pullPoolShare(msg.sender, poolAmountIn);
106         _burnPoolShare(bsub(poolAmountIn, exitFee));
107         _pushPoolShare(_factory, exitFee);
108         _pushUnderlying(tokenOut, msg.sender, tokenAmountOut);
109
110         return poolAmountIn;
111     }
```

SLOC Appendix

Solidty Contracts

Language	Files	Lines	Blanks	Comments	Code	Complexity
Solidity	7	1461	243	183	1035	54

Comments to Code 183/1035 = 18%

Javascript Tests

Language	Files	Lines	Blanks	Comments	Code	Complexity
JavaScript	7	1921	317	88	1516	28
Python	3	191	54	18	119	24
Total	10	2112	371	106	1635	52

Tests to Code 1635/1035 = 158%