Audit Report

April.13.2021

Contents

Disclaimer

This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between SHIELD and Dexon. (the\Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the\Agreement"). This Report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This Report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without SHIELD's prior written consent.

The analysis did not include any tokenomics analysis (e.g. APY rates etc.) and should not be considered as an investment advice!

Executive Summary

This report has been prepared as the product of the Smart Contract Audit request by DexonFinance. This audit was conducted to discover issues and vulnerabilities in the source code of DexonFinance's Smart Contracts. Utilizing SHIELD's Formal Verification Platform, Static Analysis, and Manual Review, a comprehensive examination has been performed. The auditing process pays special attention to the following considerations.

_ Testing the smart contracts against both common and uncommon attack vectors.
_ Assessment of the codebase for best practice and industry standards.
_ Ensuring contract logic meets the specifications and intentions of the client.
_ Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
_ Thorough line by line manual review of the entire codebase by industry experts.

Vulnerability Classification

For every issue found, SHIELD categorizes them into 3 buckets based on its risk level:

High

The code implementation does not match the speciation, or it could result in loss of funds for contract owner or users.

Medium

The code implementation does not match the speciation at certain conditions, or it could affect the security standard by lost of access control.

Low

The code implementation is not a best practice, or use a suboptimal design pattern, which may lead to security vulnerabilies, but no concern found yet.

Project Summary

Project Name	DexonFinance
Platform	Bscscan, Solidity
Github	https://github.com/Dexonfinance/solidity
Commit	0x5f50bf5106f64bfe6b71a4971cac24c10024d2ce

Manual Review Summary

Scope of Work

To ensure comprehensive protection, the source code has been analyzed by the proprietary SHIELD manually reviewed by our blockchain and smart contract experts and engineers.

That end-to-end process ensures proof of stability as well as a hands-on, engineering-focused process to close potential loopholes and recommend design changes in accordance with the best practices in the space.

Manual Checks

StratX params:

4. buyBackRate

150 *uint256*

5. buyBackRateMax

10000 *uint256*

6. buyBackRateUL

800 *uint256*

7. controllerFee

20 *uint256*

8. controllerFeeMax

10000 *uint256*

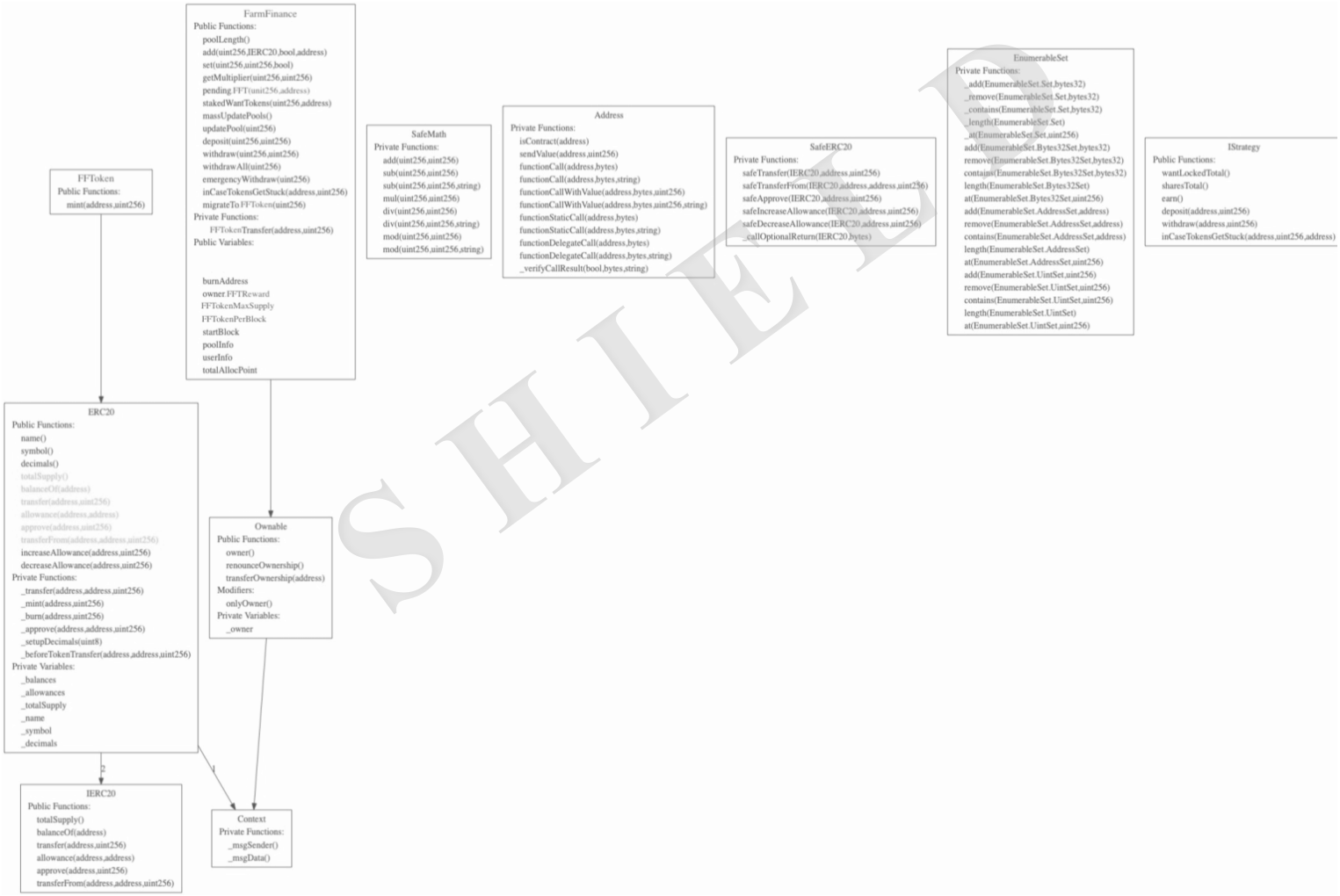
9. controllerFeeUL

300 *uint256*

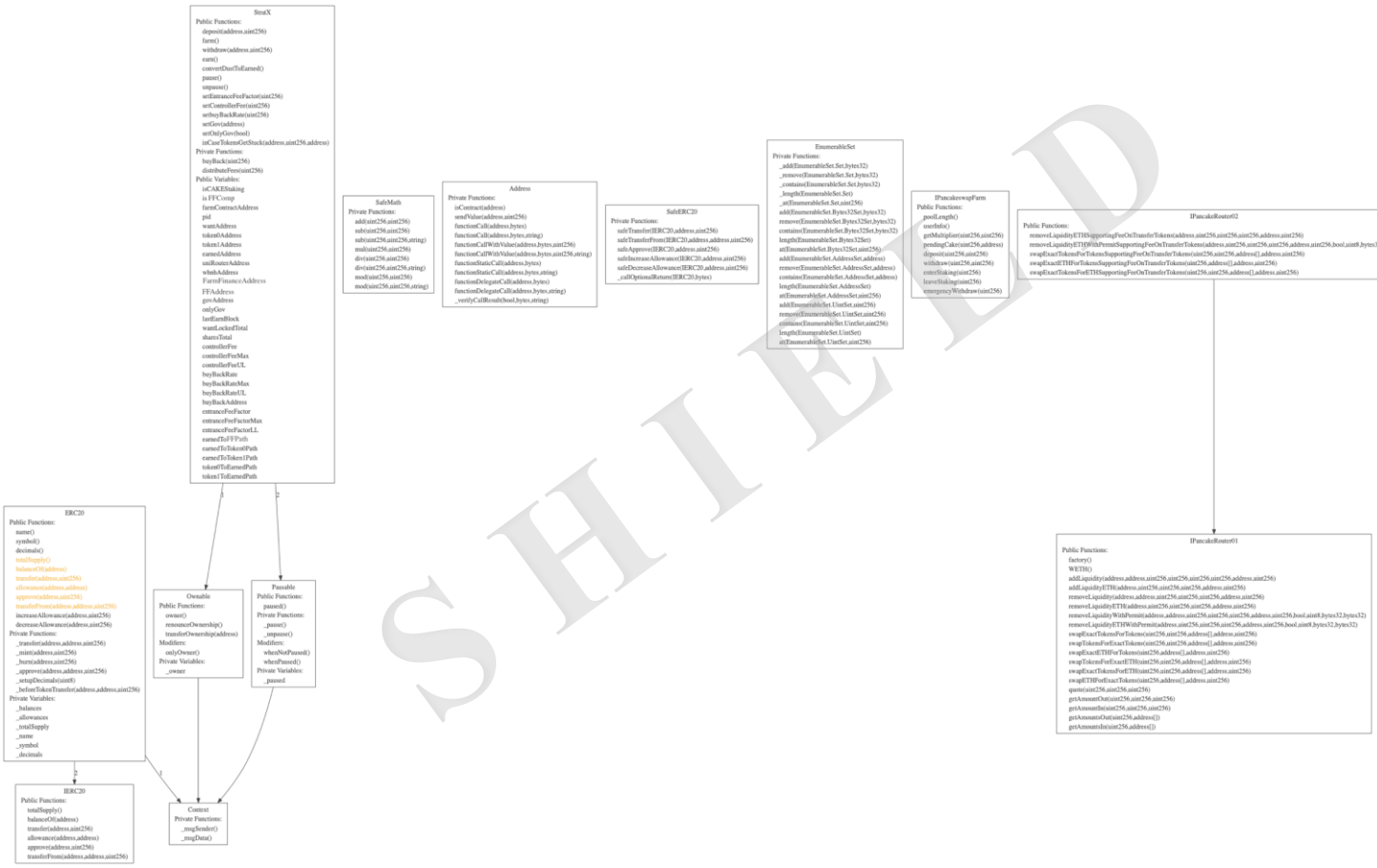
Architecture

Please find below the calling architecture of the reviewed contracts.

DexonFinance:



StratX (first strategy)



Issues

Number of contracts: 13+15+11 (including inherited ones)

```
/**
 *Submitted for verification at BscScan.com on 2021-02-25
*/

// File: contracts/lib/SafeMath.sol

/*

    Copyright 2020 DODO ZOO.
    SPDX-License-Identifier: Apache-2.0

*/

pragma solidity 0.6.9;


/**
 * @title SafeMath
 * @author DODO Breeder
 *
 * @notice Math operations with safety checks that revert on error
 */
library SafeMath {
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "MUL_ERROR");

        return c;
    }
```

Medium Issues

Divide before multiply:

Solidity integer division might truncate. It may happen that performing multiplication before division might reduce precision. [Recommendation] All those operation have
to be carefully analyzed for any precision reduction based on the code business logic. Not a significant risk.

Possible re-entrancy:

```
function transferFrom(
    address from,
    address to,
    uint256 amount
) public returns (bool) {
    require(to != address(0), "TO_ADDRESS_IS_EMPTY");
    require(amount <= balances[from], "BALANCE_NOT_ENOUGH");
    require(amount <= allowed[from][msg.sender], "ALLOWANCE_NOT_ENOUGH");

    balances[from] = balances[from].sub(amount);
    balances[to] = balances[to].add(amount);
    allowed[from][msg.sender] = allowed[from][msg.sender].sub(amount);
    emit Transfer(from, to, amount);
    return true;
}

function approve(address spender, uint256 amount) public returns (bool) {
    allowed[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}
```

Low issues:

Different versions of solidity used.

[Recommendation] Stick to one version of Solidity.

Dynamic Tests

We have run fuzzing/property-based testing of Solidity smarts contracts. It was using sophisticated grammar-based fuzzing campaigns based on a contract ABI to falsify user-defined predicates or Solidity assertions. There were also dynamic tests run on EVM byte code to detect common vulnerabilities including integer underflows, owner-overwrite-to-BNB-withdrawal, and others.

```
function transferFrom(
    address from,
    address to,
    uint256 amount
) public returns (bool) {
    require(to != address(0), "TO_ADDRESS_IS_EMPTY");
    require(amount <= balances[from], "BALANCE_NOT_ENOUGH");
    require(amount <= allowed[from][msg.sender], "ALLOWANCE_NOT_ENOUGH");

    balances[from] = balances[from].sub(amount);
    balances[to] = balances[to].add(amount);
    allowed[from][msg.sender] = allowed[from][msg.sender].sub(amount);
    emit Transfer(from, to, amount);
    return true;
}

function approve(address spender, uint256 amount) public returns (bool) {
    allowed[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}
```

Manual Check: IStrategy contract is fully controlled by the codebase, providing proper tests are done it is NOT an issue.

Automatic Tests

The project lacks any automatic testing and tests scripts. We did not run any functional tests provided by the team, due to lack of such scripts provided. Hence the full business logic functionality was not tested.

[Recommendation]: Create comprehensive test cases and implement them as scripts or mocha tests using the hardhat infrastructure.

[Disclaimer] There were no tests conducted testing full system functionality due to lack of proper test cases and/or test scripts.

Deployment & Contract Ownership

The contracts are currently deployed on Heco Mainnet.
DexonFinance: `0x5f50BF5106f64BFe6b71a4971CaC24C10024d2CE`
BUSD-DON-Strategy: `0x1ab4D2091de991eA2D632AA6B296a206579dC800`

TimelockController controls multiple functions with significant

importance to the
overall system. Currently the delay in the time lock is set to 60/30 seconds.

Conclusion

Overall we found the smart contracts to follow good practices. With the final update of source code and delivery of the audit report, we conclude that the contract is structurally sound and not vulnerable to any classically known anti-patterns or security issues. The audit report itself is not necessarily a guarantee of correctness or trustworthiness, and we always recommend to seek multiple opinions, keep improving the codebase, and more test coverage and sandbox deployments before the release.