SMART CONTRACT AUDIT REPORT

for

DEFIDOLLAR

Prepared By: Shuxiao Wang

Hangzhou, China

October 23, 2020

Document Properties

Client	DefiDollar
Title	Smart Contract Audit Report
Target	Liquidity Mining Genesis
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 23, 2020	Xuxian Jiang	Final Release
0.2	October 20, 2020	Xuxian Jiang	Additional Findings
0.1	October 18, 2020	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of **DefiDollar Liquidity Mining Genesis**, we in the report outline our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About DefiDollar

DefiDollar (DUSD) is an index of stable coins that uses DeFi primitives to stay near the dollar mark more robustly than each individual stable coin. The vision behind DUSD is to provide an avenue for diversifying users' crypto-dollars positions, and to dampen the potentially disastrous effects of a particular stable coin such as Tether failing (partially or completely) from its peg. The Liquidity Mining Genesis is in essence a genesis ceremony for the launch of its governance token, i,e., DFD.

The basic information of DefiDollar Liquidity Mining Genesis is as follows:

Table 1.1: Basic Information of DefiDollar Liquidity Mining Genesis

Item	Description
Issuer	DefiDollar
Website	https://www.defidollar.xyz/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 23, 2020

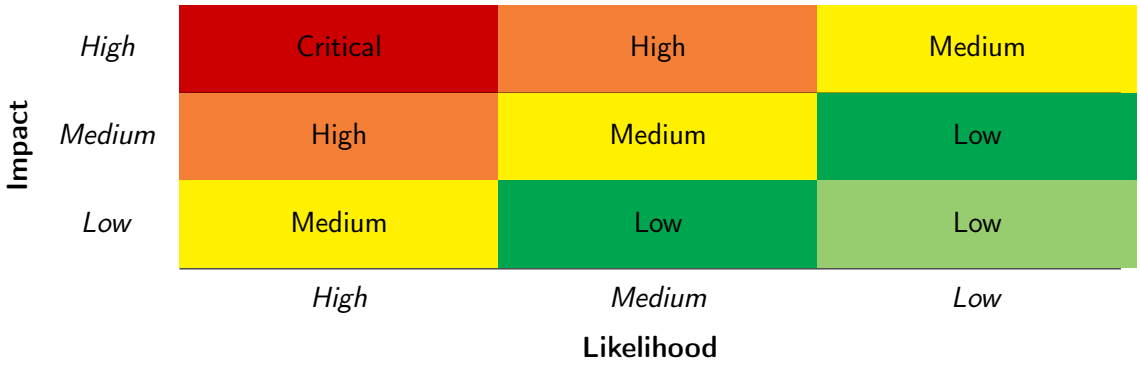
In the following, we show the repository of the reviewed code used in this audit.

- https://github.com/defidollar/genesis.git (77dcbab)

1.2 About PeckShield

PeckShield Inc. [16] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [11]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of DefiDollar Liquidity Mining Genesis. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	2	■■
Informational	2	■■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Functionality Separation Between Proxy And Logic	Coding Practices	Fixed
PVE-002	Informational	Suggested Adherence of Checks-Effects-Interactions	Time and State	Fixed
PVE-003	Low	Better Overflow Mitigation in rewardPerToken()	Numeric Errors	Fixed
PVE-004	Informational	Suggested Enforcement of DFD Balance at Launch	Business Logics	Fixed
PVE-005	Medium	Suggested Cap of DFD totalSupply	Business Logics	Partially Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Functionality Separation Between Proxy And Logic

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Genesis`, `DFDRewards`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

Ethereum smart contracts are typically immutable by default. Once they are created, there is no way to alter them, effectively acting as an unbreakable contract among participants. In the meantime, there are several scenarios where there is a need to upgrade the contracts, either to add new functionalities or mitigate potential bugs.

The upgradeability support comes with a few caveats. One important caveat is related to the initialization of new contracts that are just deployed to replace old contracts. Due to the inherent requirement of any proxy-based upgradeability system, no constructors can be used in upgradeable contracts. This means we need to change the constructor of a new contract into a regular function (typically named `initialize()`) that basically executes all the setup logic.

However, a follow-up caveat is that during a contract's lifetime, its constructor is guaranteed to be called exactly once (and it happens at the very moment of being deployed). But a regular function may be called multiple times! In order to ensure that a contract will only be initialized once, we need to guarantee that the chosen `initialize()` function can be called only once during the entire lifetime. This guarantee is typically implemented as a modifier named `initializer`.

The `Genesis` contract implements the logic functionality that is deployed behind a proxy. To facilitate our discussion, we show the code snippet of `Genesis` below. We notice that it is currently defined as a proxy that unnecessarily brings confusions on its role and the distinction from the true proxy contract.

```
10 import {UpgradableProxy} from "./proxy/UpgradableProxy.sol";

12 contract Genesis is UpgradableProxy {
13     using SafeERC20 for IERC20;
14     using SafeMath for uint;
15     using Math for uint;

17     uint public totalSupply;
18     mapping(address => uint) internal _balances;

20     uint public constant vestingDuration = 30 days;

22     IERC20 public dfd;
23     IERC20 public dusd;
24     ...
25 }
```

Listing 3.1: DFDRewards.sol

Recommendation Make a clear distinction between proxy and logic contracts and introduce no confusions for future upgrades. Note that revision needs to make the same storage layout between the proxy and the logic. Also, the unused import statement (at line 10) can be safely removed.

Status This issue has been fixed in this commit: c951782bb7e5af652575229b5af8da6d09d63772.

3.2 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DFDRewards, Genesis`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [20] exploit, and the recent `Uniswap/Lendf.Me` hack [17].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `DFDRewards` as an example, the `stake()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 241) starts before effecting the update on internal states (lines 243−243), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching `re-entrancy` via the very same `stake()` function.

```
239     function stake(uint amount) public updateReward(msg.sender) {
240         require(amount > 0, "Cannot stake 0");
241         IERC20(address(bPool)).safeTransferFrom(msg.sender, address(this), amount);
242         totalSupply = totalSupply.add(amount);
243         _balances[msg.sender] = _balances[msg.sender].add(amount);
244     }
```

Listing 3.2: DFDRewards.sol

In the meantime, we should mention that the `Balancer`'s LP tokens implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Other similar violations can be found in `participateOnBehalfOf()` (in the `Genesis` contract) and `notifyRewardAmount()` (in the `DFDRewards` contract).

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice. An example revision can be found as follows:

```
239     function stake(uint amount) public updateReward(msg.sender) {
240         require(amount > 0, "Cannot stake 0");
241         totalSupply = totalSupply.add(amount);
242         _balances[msg.sender] = _balances[msg.sender].add(amount);
243         IERC20(address(bPool)).safeTransferFrom(msg.sender, address(this), amount);
244     }
```

Listing 3.3: DFDRewards.sol (revised)

Status This issue has been fixed in this commit: c951782bb7e5af652575229b5af8da6d09d63772.

3.3 Better Overflow Mitigation in rewardPerToken()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: DFDRewards
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

The DefiDollar Liquidity Mining Genesis support defines unique incentive mechanisms to encourage early adoption. To facilitate the distribution of the governance token, i.e., DFD, in a secure manner, Liquidity Mining Genesis introduces a new contract named DFDRewards, which defines the main notifier routine of reward amounts to the incentivized pool, i.e., notifyRewardAmount().

```
282     function notifyRewardAmount(uint256 reward, uint256 duration)
283         external
284         onlyOwner
285         updateReward(address(0))
286     {
287         dfd.safeTransferFrom(msg.sender, address(this), reward);
288         uint _now = _timestamp();
289         if (_now >= periodFinish) {
290             rewardRate = reward.div(duration);
291         } else {
292             uint256 remaining = periodFinish.sub(_now);
293             uint256 leftover = remaining.mul(rewardRate);
294             rewardRate = reward.add(leftover).div(duration);
295         }
296         lastUpdateTime = _now;
297         periodFinish = _now.add(duration);
298         emit RewardAdded(reward);
299     }
```

Listing 3.4: DFDRewards.sol

To elaborate, we show the notifier logic above. This logic is restricted to owner only (with the onlyOwner modifier enforcement) and takes two arguments: reward and duration. Apparently, it intends to conveniently notify the reward pool with the reward amount for the specified duration. For safety, the notified reward amount is no larger than the sender's balance. By doing so, it can greatly alleviate the concern on the potential bug that may lock user stakes if the notified reward amount is larger enough to always trigger the RewardPool SafeMath issue [19] and thus cause revert!

Specifically, this issue stems from the calculation of rewardPerToken() function (lines 225 – 237). If a large number of reward amount is being notified, we will obtain a large rewardRate, which causes the following math lastTimeRewardApplicable().sub(lastUpdateTime).mul(rewardRate).mul(1e18).div(

_totalSupply) to overflow. Since rewardPerToken() is always invoked when stakers attempt to retrieve back their stakes, the always-reverted execution effectively blocks the attempt and thus locks the funds.

```
225     function rewardPerToken() public view returns (uint) {
226         if (totalSupply == 0) {
227             return rewardPerTokenStored;
228         }
229         return
230             rewardPerTokenStored.add(
231                 lastTimeRewardApplicable()
232                     .sub(lastUpdateTime)
233                     .mul(rewardRate)
234                     .mul(1e18)
235                     .div(totalSupply)
236             );
237     }
```

Listing 3.5: DFDRewards.sol

The current implementation essentially restricts the transferred amount to the sender's balance (line 287). This greatly alleviates the above overflow concern. To better mitigate possible overflows, it is suggested to completely eliminate the risk by explicitly validating the no overflow would ever occur.

As a solution, the revised notifyRewardAmount() routine needs to guarantee that the reward amount stays within a normal range without overflowing the rewardPerToken() calculation.

Recommendation Validate the new reward amount will not lead to an overflow in rewardPerToken(). An example revision is shown below.

```
282     function notifyRewardAmount(uint256 reward, uint256 duration)
283         external
284         onlyOwner
285         updateReward(address(0))
286     {
287         uint _now = _timestamp();
288         if (_now >= periodFinish) {
289             rewardRate = reward.div(duration);
290         } else {
291             uint256 remaining = periodFinish.sub(_now);
292             uint256 leftover = remaining.mul(rewardRate);
293             rewardRate = reward.add(leftover).div(duration);
294         }
295         lastUpdateTime = _now;
296         periodFinish = _now.add(duration);
297
298         require(rewardRate.mul(duration) < uint256(-1) / 10**18, "large rewards would
                lock");
299         dfd.safeTransferFrom(msg.sender, address(this), reward);
300         emit RewardAdded(reward);
```

```
301     }
```

Listing 3.6: DFDRewards.sol

Status This issue has been fixed in this commit: c951782bb7e5af652575229b5af8da6d09d63772.

3.4 Suggested Enforcement of DFD Balance at Launch

- ID: PVE-004
- Severity: Infomational
- Likelihood: N/A
- Impact: N/A
- Target: `Genesis`
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.3, the DefiDollar Liquidity Mining Genesis support defines unique incentive mechanisms to encourage early adoption. It is mentioned in the design document that "*Team deposits 3mil (3% supply) DFD tokens in a contract. This is not for a token sale. This is a new approach to liquidity mining.*"

With that, it is suggested to enforce the presence of the initial `DFD` balance of 3mil when it is being launched. This is certainly helpful to be consistent between the original design and its current implementation. It will also be helpful to boost the community confidence.

```
74      function launch(uint dfdDenorm, uint dusdDenorm, uint swapFee)
75          external
76          onlyOwner
77      {
78          require(_timestamp() >= liftOff, "!liftOff");
79          require(address(bPool) == address(0x0), "Already lifted off");
80
81          // instantiate a new BPool
82          bPool = bFactory.newBPool();
83
84          uint _dfd = dfd.balanceOf((address(this)));
85          uint _dusd = dusd.balanceOf((address(this)));
86          dfd.safeApprove(address(bPool), _dfd);
87          dusd.safeApprove(address(bPool), _dusd);
88
89          /*
90          Adding all liquidity all at once will reap only 100 BPTs.
91          Which will mean that each BPT represents a large number of tokens and will make
                for bad UX for users.
92          Instead we start the pool with just 100 dusd and then add liquidity to the pool.
```

```
93          Consequently, we will end up with the exact same number of BPTs as DUSD
                contributed during genesis.
94          */
95
96          // bind dfd corresponsing to 100 dusd
97          bPool.bind(address(dfd), _dfd.mul(1e20).div(_dusd), dfdDenorm);
98          // bind 100 dusd
99          bPool.bind(address(dusd), 1e20, dusdDenorm);
100
101         bPool.setSwapFee(swapFee);
102         bPool.finalize();
103
104         require(
105             bPool.balanceOf(address(this)) == 1e20,
106             "Unexpected initialization"
107         );
108
109         // mint exact number # of bPool tokens
110         uint poolAmountOut = _dusd.sub(1e20);
111         uint[] memory maxAmountsIn = new uint[](2);
112         maxAmountsIn[0] = dfd.balanceOf(address(this));
113         maxAmountsIn[1] = dusd.balanceOf(address(this));
114         bPool.joinPool(poolAmountOut, maxAmountsIn);
115
116         // gulp if/what is left into the bpool, this doesn't mint new BPTs
117         uint residue = dfd.balanceOf((address(this)));
118         if (residue > 0) {
119             dfd.safeTransfer(address(bPool), residue);
120             bPool.gulp(address(dfd));
121         }
122         residue = dusd.balanceOf(address(this));
123         if (residue > 0) {
124             dusd.safeTransfer(address(bPool), residue);
125             bPool.gulp(address(dusd));
126         }
127
128         totalSupply = bPool.balanceOf(address(this));
129         require(totalSupply == _dusd, "Unexpected initialization:2");
130     }
```

Listing 3.7: Genesis.sol

Recommendation Enforce the intended DFD balance at launch as follows:

```
74      function launch(uint dfdDenorm, uint dusdDenorm, uint swapFee)
75          external
76          onlyOwner
77      {
78          require(_timestamp() >= liftOff, "!liftOff");
79          require(address(bPool) == address(0x0), "Already lifted off");
80
81          // instantiate a new BPool
82          bPool = bFactory.newBPool();
```

```
83
84          uint _dfd = dfd.balanceOf((address(this)));
85          uint _dusd = dusd.balanceOf((address(this)));
86
87          require( _dfd == 3*1e24, "launch: initial DFD balance off");
88
89          dfd.safeApprove(address(bPool), _dfd);
90          dusd.safeApprove(address(bPool), _dusd);
91
92          /*
93          Adding all liquidity all at once will reap only 100 BPTs.
94          Which will mean that each BPT represents a large number of tokens and will make
                for bad UX for users.
95          Instead we start the pool with just 100 dusd and then add liquidity to the pool.
96          Consequently, we will end up with the exact same number of BPTs as DUSD
                contributed during genesis.
97          */
98
99          // bind dfd corresponsing to 100 dusd
100         bPool.bind(address(dfd), _dfd.mul(1e20).div(_dusd), dfdDenorm);
101         // bind 100 dusd
102         bPool.bind(address(dusd), 1e20, dusdDenorm);
103
104         bPool.setSwapFee(swapFee);
105         bPool.finalize();
106
107         require(
108             bPool.balanceOf(address(this)) == 1e20,
109             "Unexpected initialization"
110         );
111
112         // mint exact number # of bPool tokens
113         uint poolAmountOut = _dusd.sub(1e20);
114         uint[] memory maxAmountsIn = new uint[](2);
115         maxAmountsIn[0] = dfd.balanceOf(address(this));
116         maxAmountsIn[1] = dusd.balanceOf(address(this));
117         bPool.joinPool(poolAmountOut, maxAmountsIn);
118
119         // gulp if/what is left into the bpool, this doesn't mint new BPTs
120         uint residue = dfd.balanceOf((address(this)));
121         if (residue > 0) {
122             dfd.safeTransfer(address(bPool), residue);
123             bPool.gulp(address(dfd));
124         }
125         residue = dusd.balanceOf(address(this));
126         if (residue > 0) {
127             dusd.safeTransfer(address(bPool), residue);
128             bPool.gulp(address(dusd));
129         }
130
131         totalSupply = bPool.balanceOf(address(this));
132         require(totalSupply == _dusd, "Unexpected initialization:2");
```

```
133     }
```

Listing 3.8: Genesis.sol

Status This issue has been fixed in this commit: c951782bb7e5af652575229b5af8da6d09d63772.

3.5 Suggested Cap of DFD totalSupply

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: DFD
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

The governance token, DFD, is intended to be fairly distributed to the DefiDollar community and it is designed to be capped. Specifically, the initial bootstrap of the liquidity pool in Balancer consumes 3 million, which occupies 3% of its total supply. With that, we suggested to cap the total supply of DFD to better contain the token owner privilege and mitigate any unnecessary abuse risks.

```
6   contract DFD is ERC20, ERC20Detailed {
7       address public governance;
8       mapping (address => bool) public minters;
9
10      constructor () public ERC20Detailed("DefiDollarGov", "DFD", 18) {
11          governance = msg.sender;
12      }
13
14      function mint(address account, uint amount) public {
15          require(minters[msg.sender], "!minter");
16          _mint(account, amount);
17      }
18
19      function setGovernance(address _governance) public {
20          require(msg.sender == governance, "!governance");
21          governance = _governance;
22      }
23
24      function addMinter(address _minter) public {
25          require(msg.sender == governance, "!governance");
26          minters[_minter] = true;
27      }
28
29      function removeMinter(address _minter) public {
30          require(msg.sender == governance, "!governance");
31          minters[_minter] = false;
```

```
32      }
33  }
```

Listing 3.9: DFD.sol

Recommendation Ensure the totalSupply of `DFD` to be capped at 100 million. An example enforcement is shown below:

```
6   contract DFD is ERC20, ERC20Detailed, ERC20Capped {
7       address public governance;
8       mapping (address => bool) public minters;

10      constructor () public ERC20Capped(1 * 10**8 * 10**18), ERC20Detailed("DefiDollarGov"
            , "DFD", 18) {
11          governance = msg.sender;
12      }

14      function mint(address account, uint amount) public {
15          require(minters[msg.sender], "!minter");
16          _mint(account, amount);
17      }

19      function setGovernance(address _governance) public {
20          require(msg.sender == governance, "!governance");
21          governance = _governance;
22      }

24      function addMinter(address _minter) public {
25          require(msg.sender == governance, "!governance");
26          minters[_minter] = true;
27      }

29      function removeMinter(address _minter) public {
30          require(msg.sender == governance, "!governance");
31          minters[_minter] = false;
32      }
33  }
```

Listing 3.10: DFD.sol (revised)

Status This issue has been confirmed. After discussion, the team considers it is not possible yet to decide on a cap for the time being. However it has been agreed to place the `mint()` functionality behind a timelock-ed multisig.

4 Conclusion

In this audit, we have analyzed the design and implementation of DefiDollar Liquidity Mining Genesis. The proposed DefiDollar system for stable coin index presents a unique innovation and we are really impressed by the overall design and implementation. The audited liquidity mining support is helpful to bootstrap the distribution of its governance tokens. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [12, 13, 14, 15, 18].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [21] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[10] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[11] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[12] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[13] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[14] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[15] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

[16] PeckShield. PeckShield Inc. https://www.peckshield.com.

[17] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[18] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[19] Synthetix Improvement Proposals. SIP 77: StakingRewards bug fix's and Pausable stake(). https://sips.synthetix.io/sips/sip-77.

[20] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.

[21] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.