SMART CONTRACT AUDIT REPORT

for

INTEREST-BEARING DEFIDOLLAR

Prepared By: Shuxiao Wang

Hangzhou, China

December 21, 2020

Document Properties

Client	DefiDollar
Title	Smart Contract Audit Report
Target	Interest-Bearing DefiDollar
Version	1.0
Author	Xuxian Jiang
Auditors	Xudong Shao, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 21, 2020	Xuxian Jiang	Final Release
1.0-rc	December 18, 2020	Xuxian Jiang	Release Candidate
0.1	December 16, 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 **Interest-Bearing DefiDollar**, we outline in the report 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 is well-designed and engineered. This document outlines our audit results.

1.1 About Interest-Bearing 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. With yield-generating underlying protocol integrations, the audited system on Interest-Bearing DefiDollar makes a step further by presenting a powerful savings account that will be well-received by current DefiDollar community.

The basic information of Interest-Bearing DefiDollar is as follows:

Table 1.1: Basic Information of Interest-Bearing DefiDollar

Item	Description
Issuer	DefiDollar
Website	https://www.defidollar.xyz/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 21, 2020

In the following, we show the Git repositories of reviewed files and the commit hash values used

in this audit. Note this repository contains a number of sub-directories (e.g., `base`, `peaks`, and `valley`) and this audit covers only the `stream` sub-directory.

- https://github.com/defidollar/defidollar-core.git (0dc7ea9)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/defidollar/defidollar-core.git (d43e948)

1.2 About PeckShield

PeckShield Inc. [9] 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

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

- 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.

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

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.

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) [7], 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 Interest-Bearing DefiDollar. 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	1	■
Informational	2	■■
Total	4	

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, 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Improved Logic in DFD-Comptroller::harvest()	Coding Practices	Resolved
PVE-002	Low	Possible Front-Running For Reduced Return	Time and State	Resolved
PVE-003	Medium	Revisited Trust on Admin Keys	Security Features	Confirmed
PVE-004	Informational	Unused Code Removal	Coding Practices	Resolved

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 Improved Logic in DFDComptroller::harvest()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DFDComptroller`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The rewards collected in `DFDComptroller` are in the form of `DUSD` and are designed to swap for `DFD` as interests for staking `DFD` holders. Our examination of the `DUSD -> DFD` conversion process shows that the logic can be improved.

To illustrate, we show below the `harvest()` routine in `DFDComptroller`. This routine is tasked with collecting the `DUSD` rewards from the `comptroller` and converting the rewards to `DFD`. The conversion relies on the popular AMM-based `UniswapV2`. However, the way to interact with `UniswapV2` can be improved.

```
110     function harvest()
111         onlyOwner
112         external
113     {
114         // This contract will receive dusd because it should be a registered beneficiary
115         comptroller.harvest();
116
117         uint256 _dusd = dusd.balanceOf(address(this));
118         if (_dusd > 0) {
119             dusd.approve(uni, _dusd);
120
121             address[] memory path = new address[](3);
122             path[0] = address(dusd);
123             path[1] = address(dfd);
124
125             uint[] memory amounts = Uni(uni).swapExactTokensForTokens(_dusd, uint256(0),
                    path, address(this), now.add(1800));
```

```
126             if (amounts[1] > 0) {
127                 dfd.safeTransfer(beneficiary, amounts[1]);
128             }
129             emit Harvested(_dusd, amounts[1]);
130         }
131     }
```

Listing 3.1: DFDComptroller::harvest()

Specifically, we notice the conversion path is initialized with an array with three elements (line 121) with the first two elements as DUSD and DFD respectively. In fact, the conversion path only needs two elements, not three.

Recommendation Revise the `harvest()` logic to correct the array size. An example revision is shown below.

```
110     function harvest()
111         onlyOwner
112         external
113     {
114         // This contract will receive dusd because it should be a registered beneficiary
115         comptroller.harvest();
116 
117         uint256 _dusd = dusd.balanceOf(address(this));
118         if (_dusd > 0) {
119             dusd.approve(uni, _dusd);
120 
121             address[] memory path = new address[](2);
122             path[0] = address(dusd);
123             path[1] = address(dfd);
124 
125             uint[] memory amounts = Uni(uni).swapExactTokensForTokens(_dusd, uint256(0),
                    path, address(this), now.add(1800));
126             if (amounts[1] > 0) {
127                 dfd.safeTransfer(beneficiary, amounts[1]);
128             }
129             emit Harvested(_dusd, amounts[1]);
130         }
131     }
```

Listing 3.2: DFDComptroller::harvest()

Status This issue has been fixed in this commit: 99e2dc3c.

3.2 Possible Front-Running For Reduced Return

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: DFDComptroller
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

In Interest-Bearing DefiDollar, the rewards are accumulated from a number of sources, including the trading fees generated on the Curve pools, the yield from the yUSD Peak of DefiDollar, the DUSD redeem fee (0.1%), and the ibDUSD redemption fee (0.5%).

Using the DFDComptroller contract as an example, the authorized owner can call harvest() that basically collects any pending rewards (via comptroller.harvest() - line 115), swaps for the designated DFD (line 125), and returns the rewards to its beneficiary (line 127).

```
110     function harvest()
111         onlyOwner
112         external
113     {
114         // This contract will receive dusd because it should be a registered beneficiary
115         comptroller.harvest();
116
117         uint256 _dusd = dusd.balanceOf(address(this));
118         if (_dusd > 0) {
119             dusd.approve(uni, _dusd);
120
121             address[] memory path = new address[](3);
122             path[0] = address(dusd);
123             path[1] = address(dfd);
124
125             uint[] memory amounts = Uni(uni).swapExactTokensForTokens(_dusd, uint256(0),
                    path, address(this), now.add(1800));
126             if (amounts[1] > 0) {
127                 dfd.safeTransfer(beneficiary, amounts[1]);
128             }
129             emit Harvested(_dusd, amounts[1]);
130         }
131     }
```

Listing 3.3: DFDComptroller::harvest()

We notice the collected yields are routed to UniswapV2 in order to swap them to DFD as rewards. And the swap operation does not specify any restriction on possible slippage and is vulnerable to possible front-running attacks, possibly resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user or the `beneficiary` in our case because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status This issue has been confirmed and fixed in this commit by passing in minimal expected amount as an argument to `harvest()`: 99e2dc3c.

3.3 Revisited Trust on Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ibDFD, ibDUSD`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In Interest-Bearing DefiDollar, there is a privileged contract, i.e., `owner`, that plays a critical role in accepting (and returning) assets from (and to) staking users. This contract is designed to greatly facilitate the interest generation, collection, and distribution for depositing stakers.

In the following, we use the `ibDUSD` contract as an example and show below the code snippets of two routines, i.e., `deposit()` and `withdraw()`. The first routine is used to accept user deposits and calculate the corresponding share on the pool while the second routine computes the withdraws when given the pool share. Note that the withdrawal amount will be deducted by the so-called `redemption fee`. This `redemption fee` is specified by a system-wide risk parameter – `redeemFactor` (line 48).

```
33      function deposit(uint _amount) external harvest {
34          uint _pool = balance();
35          dusd.safeTransferFrom(msg.sender, address(this), _amount);
36          uint shares = 0;
37          if (_pool == 0) {
38              shares = _amount;
```

```
39          } else {
40              shares = _amount.mul(totalSupply()).div(_pool);
41          }
42          _mint(msg.sender, shares);
43      }
44
45      function withdraw(uint _shares) external harvest {
46          uint r = balance()
47              .mul(_shares)
48              .mul(redeemFactor)
49              .div(totalSupply().mul(FEE_PRECISION));
50          _burn(msg.sender, _shares);
51          dusd.safeTransfer(msg.sender, r);
52      }
```

Listing 3.4: The deposit() and withdraw() functions in ibDUSD

We also notice the asset accepted for deposits and withdraws is specified in another parameter – `dusd`. Both `dusd` and `redeemFactor` can be dynamically re-configured by the privileged `owner` (via `setParams()` as shown below).

```
72      function setParams(
73          IERC20 _dusd,
74          IComptroller _controller,
75          uint _redeemFactor
76      )   external
77          onlyOwner
78      {
79          require(
80              address(_dusd) != address(0) && address(_controller) != address(0),
81              "0 address during initialization"
82          );
83          require(
84              _redeemFactor <= FEE_PRECISION,
85              "Incorrect upper bound for fee"
86          );
87          dusd = _dusd;
88          controller = _controller;
89          redeemFactor = _redeemFactor;
90      }
```

Listing 3.5: ibDUSD::setParams()

While it is necessary to have a privileged account to adjust the above risk parameters, it is worrisome that the adjustment can be done after users stake their assets. In other words, it is possible to create a scenario where the `owner` specifies a low or even 0 `redeemFactor` at the very beginning to attract holders to stake their assets and, later on (possibly after a significant number of accumulated assets), dramatically increases the `redeemFactor`. The stakers have the right to know the redemption fee in advance before making the staking decisions.

Moreover, the dynamic changes of `dusd` configuration is also worrisome as such change may possi-

bly lead to the undesirable consequence that stakers are unable to withdraw their funds. Specifically, if we consider the situation with a compromised `owner` account, the compromised owner may exploit the privilege to create a fake `dusd`, mint an extremely large amount to himself, then deposit into the `ibDUSD` contract to obtain an unreasonably large share of the pool, next change back to the normal `dusd`, and finally withdraw almost all staked funds in the pool.

Note the `ibDFD` contract shares the same issue.

Recommendation Promptly seal the `setParams()` interface to ensure the parameters will not be changed after starting to accept user deposits. Another alternative is to introduce a timelock mechanism before reflecting the changes on these parameters.

Status This issue has been confirmed.

3.4 Unused Code Removal

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ibDFD`, `ibDUSD`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

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 `ibDFD` contract implements the logic functionality that is deployed behind a proxy. To facilitate our discussion, we show the code snippet of `ibDFD` below. We notice this logic contract uses its

constructor that calls the constructor of its parent contract, i.e., `ERC20Detailed`. It comes to our attention that this constructor is considered redundant as it does not actually change any states in the proxy counterpart. From the software engineering perspective, this constructor can be considered as redundant code and can be safely removed.

```
12 contract ibDFD is OwnableProxy, Initializable, ERC20, ERC20Detailed {
13     using SafeERC20 for IERC20;
14
15     uint constant FEE_PRECISION = 10000;
16
17     IERC20 public dfd;
18     IDFDComptroller public comptroller;
19     uint public redeemFactor;
20
21     /**
22     * @dev Since this is a proxy, the values set in the ERC20Detailed constructor are
           not actually set in the main contract.
23     */
24     constructor ()
25         public
26         ERC20Detailed("ibDFD Implementation", "ibDFD_i", 18) {}
27     ...
28 }
```

Listing 3.6: ibDFD::**constructor**()

Another contract, i.e., `ibDUSD`, shares the same issue.

Recommendation Remove the unnecessary constructor in both `ibDFD` and `ibDUSD`.

Status This issue has been resolved. Due to the need to inherit from `ERC20Detailed` to keep the storage layout same and the fact that not passing values to constructor will make the contract abstract, the team decides to leave it as is.

4 Conclusion

In this audit, we have analyzed the design and implementation of Interest-Bearing DefiDollar. The underlying DefiDollar system for stable coin index presents a unique innovation. The audited Interest-Bearing DefiDollar makes a step further by presenting a powerful savings account that will be well-received by current DefiDollar community. 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.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

[6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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