SMART CONTRACT AUDIT REPORT

for

Opeth Protocol

Prepared By: Shuxiao Wang

PeckShield

June 3, 2021

Document Properties

Client	Defidollar
Title	Smart Contract Audit Report
Target	Defidollar Opeth Protocol
Version	1.0
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 3, 2021	Yiqun Chen	Final Release
0.1	May 28, 2021	Yiqun Chen	First 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 the `Defidollar` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, 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 Opeth Protocol

The `Opeth` protocol is a synthetic instrument that combines put options and the underlying assets. This combo, put options and underlying assets, has a lower bound, and can be used as collateral to issue stablecoins. The `governance` role has the responsibility to govern and regulate this combo to ensure the safety and liquidity of the assets. At the same time, the `governance` role also earns the `mintFee` from the user who chooses to put their funds in the `Opeth` contract. As a new decentralized financial product of the `Defidollar` team, the audited system on the `Opeth` protocol makes a step further by combining the varying assets as collateral, which makes it have more possibilities.

The basic information of the `Opeth` protocol is as follows:

Table 1.1: Basic Information of The `Opeth` Protocol

Item	Description
Issuer	Defidollar
Website	https://www.defidollar.xyz/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 3, 2021

In the following, we show the Git repository and the commit hash value used in this audit:

- https://github.com/defidollar/opeth/tree/0a1d18d (0a1d18d)

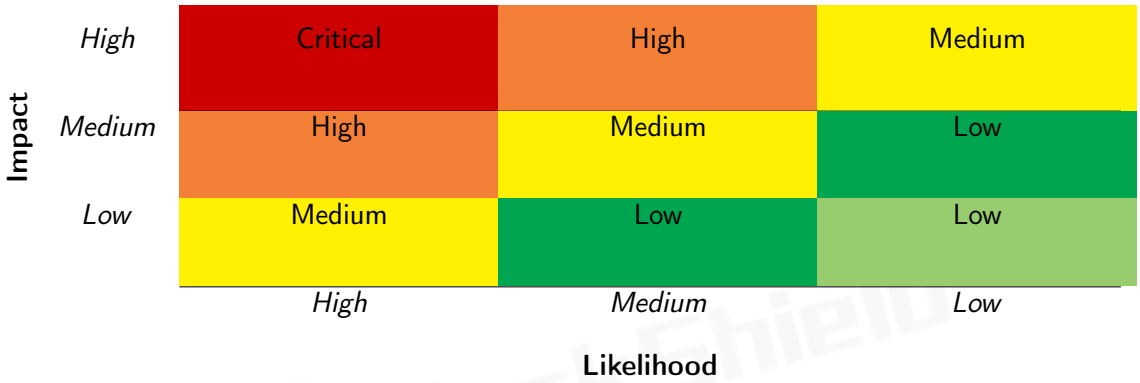
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/defidollar/opeth (f776e9b)

1.2 About PeckShield

PeckShield Inc. [6] 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 OWASP Risk Rating Methodology [5]:

- 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.

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

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) [4], 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 security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, 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 Defidollar Opeth Protocol implementation. 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	0	
Low	2	■■
Informational	0	
Total	2	

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 that 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 2 low-severity vulnerabilities.

Table 2.1: Key Defidollar Opeth Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Two-step Transfer Of Privileged Account Ownership	Coding Practices	Confirmed
PVE-002	Low	Potential Reentrancy Risk In Opeth::redeem()	Coding Practices	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Two-step Transfer Of Privileged Account Ownership

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target:
- Category: Coding Practices [3]
- CWE subcategory: CWE-561 [2]

Description

The `Defidollar` protocol implements a rather basic access control mechanism that allows a privileged account, i.e., `governance`, to be granted exclusive access to typically sensitive functions (e.g., `spawn()`). Because of the privileged access and the implications of this sensitive function, the `governance` account is essential for the protocol-level safety and operation. In the following, we elaborate with the `governance` account.

```
228     function setGovernance(address _governance) external onlyGovernance {
229         _setGovernance(_governance);
230     }

232     function _setGovernance(address _governance) internal {
233         require(_governance != address(0), "NULL_GOVERNANCE");
234         governance = _governance;
235     }
```

Listing 3.1: DSAuth::setGovernance()

The current implementation provides a specific function, i.e., `setGovernance()`, to allow for possible `governance` updates. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the new `_governance` parameter is always correctly provided. However, in the unlikely situation, when an incorrect new `_governance` is provided, the contract governance may be forever lost, which might be devastating for `Defidollar` operation and maintenance.

As a common best practice, instead of achieving the governance role update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the governance role update intent and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract governance role to an uncontrolled address. In other words, this two-step procedure ensures that a governance public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the governance role transfer process.

Recommendation Implement a two-step approach for the governance role update (or transfer): `setGovernance()` and `acceptGovernance()`.

Status This issue has been confirmed.

3.2 Potential Reentrancy Risk In Opeth::redeem()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Opeth`
- Category: Coding Practices [3]
- CWE subcategory: CWE-190 [1]

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.

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

```
135     /**
136      * @notice redeem Opeth tokens
137      * @param _amount Amount of Opeth to redeem
138      */
139     function redeem(address oToken, uint _amount) external {
140         uint _oToken = _amount.div(1e10);
```

```
141         OP20MetaData storage _op = op20s[oToken];
142         if (_op.proceedsClaimed) {
143             _processPayout(_op.collateralAsset, _op.unitPayout, _oToken);
144         } else if (isSettlementAllowed(oToken)) {
145             claimProceeds(oToken);
146             _processPayout(_op.collateralAsset, _op.unitPayout, _oToken);
147         } else {
148             // send back vanilla OTokens, because it is not yet time for settlement
149             IERC20(oToken).safeTransfer(
150                 msg.sender,
151                 _oToken
152             );
153         }
154         _op.op20.burn(msg.sender, _amount);
155         _op.underlyingAsset.safeTransfer(
156             msg.sender,
157             _oTokenToUnderlyingQuantity(_op.underlyingDecimals, _oToken)
158         );
159         emit Redeem(oToken, _amount, msg.sender);
160     }
```

Listing 3.2: Opeth::redeem()

```
162     /**
163      * @notice Redeem OTokens for payout, if any
164      */
165     function claimProceeds(address oToken) public {
166         Actions.ActionArgs[] memory _actions = new Actions.ActionArgs[](1);
167         _actions[0].actionType = Actions.ActionType.Redeem;
168         _actions[0].secondAddress = address(this);
169         _actions[0].asset = oToken;
170         _actions[0].amount = IERC20(oToken).balanceOf(address(this));

172         controller.operate(_actions);

174         OP20MetaData storage _op = op20s[oToken];
175         _op.unitPayout = MarginCalculatorInterface(addressBook.getMarginCalculator()).
                getExpiredPayoutRate(oToken);
176         _op.proceedsClaimed = true;
177         emit ClaimedProceeds(oToken);
178     }
```

Listing 3.3: Opeth::claimProceeds()

Apparently, the interaction with the external contract (line 172) starts before effecting the update on internal states (line 175), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `redeem()` function.

Recommendation Add the `nonReentrant` modifier to prevent reentrancy.

Status This issue has been addressed by the following commit: `f776e9b`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Opeth protocol. The audited contract combines put options and the underlying assets as a combo, which can be used as collateral by the user to mint stablecoins. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that Solidity-based 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[2] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

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

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

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