SMART CONTRACT AUDIT REPORT

for

DeFi Options

Prepared By: Yiqun Chen

PeckShield

September 24, 2021

Document Properties

Client	DeFiOptions
Title	Smart Contract Audit Report
Target	DeFiOptions
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 24, 2021	Xuxian Jiang	Final Release
1.0-rc2	September 23, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	September 11, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction **4**

1.1 About DeFiOptions 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Possible Flashloans on Protocol Approval/Rejection 11

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern 13

3.3 Accommodation of Non-ERC20-Compliant Tokens 14

3.4 Lack Of Duplicate Checks in setContractAddress() 16

3.5 Improved Sanity Checks in System/Function Arguments 17

3.6 Trust Issue of Admin Keys 18

3.7 Improved Logic in MoreMath::sqrt()/powDecimal() 19

3.8 Malicious Proposal Submission For Setting Changes 20

4 Conclusion **22**

References **23**

1 Introduction

Given the opportunity to review the **DeFiOptions** design document and related smart contract source code, 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 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 DeFiOptions

`DeFiOptions` is an experimental DeFi options-trading protocol that enables long and short positions for `CALL`/`PUT`-style, tokenized, collateralized, cash-settable European style options. It mainly consists of `Options Exchange` and `Credit Provider`. The `Options Exchange` accepts stablecoin deposits as collateral for issuing ERC20 option tokens. The `Credit Provider` settles or liquidates upon the maturity of each option contract. In case any option writer happens to be short on funds during settlement, the `Credit Provider` will register a debt and cover payment obligations, essentially performing a lending operation.

The basic information of DeFiOptions is as follows:

Table 1.1: Basic Information of DeFiOptions

Item	Description
Name	DeFiOptions
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 24, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/DeFiOptions/DeFiOptions-core.git (f98caf3)

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

- https://github.com/DeFiOptions/DeFiOptions-core.git (ae9cdb4)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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 Audit Checklist

Category	Checklist Items
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 checklist of 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) [11], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 Logic	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 the `DeFiOptions` protocol. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	2	■■
Low	4	■■■■
Informational	0	
Total	8	

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 2 high-severity vulnerabilities, 2 medium-severity vulnerabilities, and 4 low-severity vulnerabilities.

Table 2.1: Key DeFiOptions Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Possible Flashloans on Protocol Approval/Rejection	Business Logic	Fixed
PVE-002	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time And State	Fixed
PVE-003	Medium	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	Fixed
PVE-004	Low	Lack Of Duplicate Checks in setContractAddress()	Coding Practice	Fixed
PVE-005	Low	Improved Sanity Checks in System/Function Arguments	Coding Practice	Fixed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Low	Improved Logic in MoreMath::sqrt()/powDecimal()	Numeric Errors	Fixed
PVE-008	High	Malicious Proposal Submission For Setting Changes	Business Logic	Fixed

Besides 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 Possible Flashloans on Protocol Approval/Rejection

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Proposal`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `DeFiOptions` protocol has a built-in proposal subsystem that allows for qualified proposers to submit proposals. Users may vote on these proposals that, if passed, may make updates on various aspects of the protocol. Our analysis shows the current subsystem may be susceptible to potential flashloan attacks.

To elaborate, we show below the `castVote()` function. It takes into account the current voter balance and adds to the cumulative `yea` or `nay` votes based on the voter's choice. In the meantime, there is another public `close()` function that can be called to check whether the quorum is reached or not.

```
    function castVote(bool support) public {

        ensureIsActive();
        require(votes[msg.sender] == 0);

        uint balance = govToken.balanceOf(msg.sender);
        require(balance > 0);

        if (support) {
            votes[msg.sender] = int(balance);
            yea = yea.add(balance);
        } else {
            votes[msg.sender] = int(-balance);
            nay = nay.add(balance);
        }
```

```
    }
```

Listing 3.1: Proposal::castVote()

```
    function close() public {

        ensureIsActive();

        uint total = settings.getCirculatingSupply();

        uint v;
        if (quorum == Proposal.Quorum.SIMPLE_MAJORITY) {
            v = total.div(2);
        } else if (quorum == Proposal.Quorum.TWO_THIRDS) {
            v = total.mul(2).div(3);
        } else {
            revert();
        }

        if (yea > v) {
            status = Status.APPROVED;
            execute(settings);
        } else if (nay >= v) {
            status = Status.REJECTED;
        } else {
            revert("quorum not reached");
        }

        closed = true;
    }
```

Listing 3.2: Proposal::close()

It comes to our attention that if the `govToken` can be borrowed with a huge amount in a flashloan transaction, a malicious actor can then cast the vote to significantly increase `yea` or `nay` votes, next immediately call `close()` to pass or fail the proposal, and finally repay the flashloan.

Recommendation Revise the current voting logic to thwart the above flashloan attack.

Status The issue has been fixed by the following commits: `22558e9` and `afba27d8`.

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RedeemableToken`
- Category: Time and State [9]
- 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` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `RedeemableToken` as an example, the `redeem()` 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`. For example, the interaction with the external contract (line 53) start before effecting the update on internal states (line 54), 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 same entry function.

```
44    function redeem(uint valTotal, uint supplyTotal, address owner)
45        private
46        returns (uint bal, uint val)
47    {
48        bal = balanceOf(owner);
49
50        if (bal > 0) {
51            uint b = 1e3;
52            val = MoreMath.round(valTotal.mul(bal.mul(b)).div(supplyTotal), b);
53            exchange.transferBalance(owner, val);
54            removeBalance(owner, bal);
55        }
56
57        afterRedeem(owner, bal, val);
58    }
```

Listing 3.3: `RedeemableToken::redeem()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been fixed by this commit: `91073b5`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
```

```
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `release()` routine in the `UnderlyingVault` contract. If the `USDT` token is supported as `underlying`, the unsafe version of `IERC20(underlying).transfer(owner, value)` (line 118) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
104     function release(address owner, address token, address feed, uint value) external {

106         ensureCaller();

108         require(owner != address(0), "invalid owner");
109         require(token != address(0), "invalid token");
110         require(feed != address(0), "invalid feed");

112         uint bal = allocation[owner][token];
113         value = MoreMath.min(bal, value);

115         if (bal > 0) {
116             address underlying = UnderlyingFeed(feed).getUnderlyingAddr();
117             allocation[owner][token] = bal.sub(value);
118             IERC20(underlying).transfer(owner, value);
119             emit Release(owner, token, value);
120         }
121     }
```

Listing 3.5: UnderlyingVault :: release ()

The same issue is also present in other routines, including `depositTokens()/writeCovered()` from `OptionsExchange` and `UnderlyingVault()/release()` from `UnderlyingVault`. We highlight that the `approve()`-related idiosyncrasy needs to be addressed by applying `safeApprove()` twice: the first one reduces the allowance to 0 and the second one sets the new intended allowance.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status The issue has been fixed by this commit: `68c7a32`.

3.4 Lack Of Duplicate Checks in setContractAddress()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Deployer`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

The `DeFiOptions` protocol has a common registry contract, i.e., `Deployer`, to manage each component contract. And this registry contract provides public functions to update and retrieve key-mapped contracts.

To elaborate, we show below the related `setContractAddress()` function. As the name indicates, this function is used to set a specific active contract based on the given `key` into the registry. It comes to our attention that this routine can be improved to ensure the `key` does not duplicate existing ones. In other words, it is helpful to add another requirement `require(!has(key))` to rule out the possibility of having a duplicate key in the registry.

```
    function setContractAddress(string memory key, address addr) public {

        setContractAddress(key, addr, true);
    }

    function setContractAddress(string memory key, address addr, bool upgradeable)
        public {

        ensureNotDeployed();
        ensureCaller();

        contracts.push(ContractData(key, addr, upgradeable));
        contractMap[key] = address(1);
    }
```

Listing 3.6: Deployer::setContractAddress()

Recommendation Revise the current `setContractAddress()` function to ensure no duplicate keys exist.

Status The issue has been fixed by this commit: `d3e291e`.

3.5 Improved Sanity Checks in System/Function Arguments

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `DeFiOptions` protocol is no exception. Specifically, if we examine the `ProtocolSettings` contract, it has defined a number of protocol-wide risk parameters, e.g., `processingFee`, `circulatingSupply` and `swapTolerance`. In the following, we show an example routine that allows for their changes.

```
    function setProcessingFee(uint f, uint b) external {

        ensureWritePrivilege();
        processingFee = Rate(f, b, MAX_UINT);
    }
```

Listing 3.7: `ProtocolSettings::setProcessingFee()`

Our result shows the update logic on the above parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large `processingFee` parameter will revert every payment processing.

Moreover, there are a few functions that can benefit from improved sanity checks on the given arguments. For example, the `initializeSamples()` routine can be enhanced by requiring the two input arrays have the same length and the given `_timestamps` array contains timestamp members in the ascending order. The `getDailyVolatilityCached()` routine can be enhanced to ensure `require(period >=1)` and the `swapUnderlyingForStablecoin()` routine can be better validated with `require(path.length ==2)`.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status The issue has been fixed by the following commits: `8aa3c89`, `9baf13e`, and `0f00006`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the DeFiOptions protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and fee adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function setParameters(
        uint _spread,
        uint _reserveRatio,
        uint _withdrawFee,
        uint _mt
    )
        external
    {
        ensureCaller();
        spread = _spread;
        reserveRatio = _reserveRatio;
        withdrawFee = _withdrawFee;
        _maturity = _mt;
    }
```

Listing 3.8: LiquidityPool::setParameters()

Note that if the privileged owner account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig owner account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks.

Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed with the team. The team further clarifies the use of a multisig owner account for the `Proxy` contracts. Also, the team plans to add a timelock that will allow the `admin` account with writing privileges to the `ProtocolSettings` only for the specified days (e.g., 30 days) for performing management tasks in the initial days of operation of the protocol. After that, only the DAO will be able to change protocol settings. And a specific `setNonUpgradable()` function is now added to the `Proxy` contract so that it can be exercised to become non-upgradeable – as shown in the following commit: `354c9a8`.

3.7 Improved Logic in MoreMath::sqrt()/powDecimal()

- ID: PVE-007
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `MoreMath`
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [2]

Description

In the `DeFiOptions` protocol provides its own `MoreMath` library to facilitate a variety of arithmetic operations. Among these operations, we observe the familiar `sqrt()` function to calculate the integer square root of a given number. The `sqrt()` follows the `Babylonian` method for calculating the integer square root. Specifically, for a given x, we need to find out the largest integer z such that $z^2 <= x$.

```
86      function sqrt(uint x) internal pure returns (uint y) {
87
88          uint z = (x.add(1)).div(2);
89          y = x;
90          while (z < y) {
91              y = z;
92              z = (x.div(z).add(z)).div(2);
93          }
94      }
```

Listing 3.9: `MoreMath::sqrt()`

We show above current `sqrt()` implementation. The initial value of z to the iteration was given as $z = (x.add(1)).div(2)$ (line 88), which results in an integer overflow when $x = max\_int256 = int256(2 ** 255 - 1)$. In other words, the overflow essentially sets z to zero, leading to a `division by zero` in the calculation of $z = (x.div(z).add(z)).div(2)$ (line 92).

Note that this does not result in an incorrect return value from `sqrt()`, but does cause the function to revert unnecessarily when the above corner case occurs. Meanwhile, it is worth mentioning that if there is a `divide by zero`, the execution or the contract call will be thrown by executing the `INVALID` opcode, which by design consumes all of the gas in the initiating call. This is different from `REVERT` and has the undesirable result in causing unnecessary monetary loss. To address this particular corner case, We suggest to change the initial value to $z = (x.div(2)).add(1)$, making `sqrt()` well defined over its all possible inputs.

Recommendation Revise the above calculation to avoid the unnecessary integer overflow.

Status The issue has been fixed by this commit: `8d790ca`.

3.8 Malicious Proposal Submission For Setting Changes

- ID: PVE-008
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `GovToken`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.1, the `DeFiOptions` protocol has a built-in proposal subsystem that allows for qualified proposers to submit proposals. Users may vote on these proposals that, if passed, may make updates on various aspects of the protocol. Our analysis shows a malicious proposal can be crafted to update various protocol settings!

To elaborate, we show below the `registerProposal()` function that allows for qualified proposers to register a new proposal. Note a malicious actor can craft a new proposal, which can always be registered with a new proposal ID assigned. In other words, the call to `govToken.isRegisteredProposal(craftedProposal)` will return `true`.

```
    function registerProposal(address addr) public returns (uint id) {

        require(
            proposingDate[addr] == 0  time.getNow().sub(proposingDate[addr]) > 1 days,
            "minimum interval between proposals not met"
        );

        Proposal p = Proposal(addr);
        (uint v, uint b) = settings.getMinShareForProposal();
        require(calcShare(msg.sender, b) >= v);

        id = serial++;
```

```
        p.open(id);
        proposalsMap[id] = p;
        proposingDate[addr] = time.getNow();
        proposals.push(id);
    }
```

Listing 3.10: `GovToken::registerProposal()`

After that, the crafted proposal can have its own `isExecutionAllowed()` function that always returns `true`. Therefore, the crafted proposal can successfully pass the validation of `ensureWritePrivilege()` from the `ProtocolSettings` contract. In other words, all privileged setters are now accessible to the crafted proposal, including `setCirculatingSupply()`, `setAllowedToken()`, and `setUdlFeed()`!

```
    function ensureWritePrivilege() private view {

        if (msg.sender != owner) {
            Proposal p = Proposal(msg.sender);
            require(govToken.isRegisteredProposal(msg.sender), "proposal not registered");
            require(p.isExecutionAllowed(), "execution not allowed");
        }
    }
```

Listing 3.11: `ProtocolSettings::ensureWritePrivilege()`

Recommendation Ensure the proposal can only be submitted from a trusted entity. Or dynamically instantiate the proposal from the known implementation so that no crafted proposal may be accepted.

Status The issue has been fixed by this commit: `cfc4145`.

4 Conclusion

In this audit, we have analyzed the `DeFiOptions` design and implementation. The system presents a unique, robust offering as a decentralized options-trading protocol that enables long and short positions for `CALL`/`PUT`-style, tokenized, collateralized, cash-settable European style options. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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

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

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

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

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

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