SMART CONTRACT AUDIT REPORT

for

Hegic HardCore Beta

Prepared By: Yiqun Chen

PeckShield

February 26, 2022

Document Properties

Client	Hegic
Title	Smart Contract Audit Report
Target	Hegic HardCore Beta
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 26, 2022	Xiaotao Wu	Final Release
1.0-rc	February 22, 2022	Xiaotao Wu	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 Hegic . . . 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 Inconsistent Implementation Of _calculatePeriodFee() . . . 11

3.2 Non-Functional Logic Of HegicStakeAndCover::provide() . . . 13

3.3 Arithmetic Underflow Avoidance In HegicOperationalTreasury::_replenish() . . . 14

3.4 Incorrect lockedPremium Update In HegicOperationalTreasury::lockLiquidityFor() . . . 15

3.5 Incorrect withdraw Logic In HegicOperationalTreasury . . . 16

3.6 Accommodation of Non-ERC20-Compliant Tokens . . . 17

3.7 Trust Issue of Admin Keys . . . 19

3.8 Improved Sanity Checks For System/Function Parameters . . . 21

3.9 Improved Reentrancy Protection In HegicPool . . . 23

4 Conclusion **25**

References **26**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Hegic` protocol, 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 Hegic

The `Hegic` protocol is an on-chain peer-to-pool options trading protocol built on `Ethereum`. It works like an `AMM` (automated market maker) for options. Users can trade non-custodial on-chain call and put options as an individual holder using the simplest and intuitive interfaces. The protocol allows for the use of `MetaMask`, `Trust Wallet` or `Argent` wallets to trade options without `KYC`, email or registration required. The `Hegic` protocol provides a valuable instrument to hedge risks and control excessive exposure from market fluctuation and dynamics, therefore presenting a unique contribution to current `DeFi` ecosystem. The basic information of `Hegic` is as follows:

Table 1.1: Basic Information of Hegic HardCore Beta

Item	Description
Target	Hegic HardCore Beta
Website	https://www.hegic.co/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 26, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited repository contains a number of sub-directories and this audit only covers the `Options` and `Pool` sub-directories.

- https://github.com/hegic/hegic-hardcore-beta.git (e22e6f6)

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

- https://github.com/hegic/hegic-hardcore-beta.git (9387bbc)

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 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.

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) [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.

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 design and implementation of the `Hegic` protocol smart contracts. 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	1	■
Medium	2	■ ■
Low	4	■ ■ ■ ■
Informational	1	■
Undetermined	1	■
Total	9	

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 high-severity vulnerability, 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Inconsistent Implementation Of _calculatePeriodFee()	Coding Practices	Resolved
PVE-002	Low	Non-Functional Logic Of HegicStakeAndCover::provide()	Business Logic	Resolved
PVE-003	Low	Arithmetic Underflow Avoidance In HegicOperationalTreasury::_replenish()	Numeric Errors	Resolved
PVE-004	High	Incorrect lockedPremium Update In HegicOperationalTreasury::lockLiquidityFor()	Business Logic	Resolved
PVE-005	Medium	Incorrect withdraw Logic In HegicOperationalTreasury	Business Logic	Resolved
PVE-006	Informational	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-008	Low	Improved Sanity Checks For System/-Function Parameters	Coding Practices	Resolved
PVE-009	Undetermined	Improved Reentrancy Protection In HegicPool	Time and State	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 Inconsistent Implementation Of _calculatePeriodFee()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: PriceCalculator/PriceCalculatorUtilization
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

In the Hegic protocol, both the PriceCalculator contract and the PriceCalculatorUtilization contract implement the _calculatePeriodFee() function to calculate and price the time value of an option. While reviewing the implementations of the _calculatePeriodFee() routine in both contract, we notice that there exists certain inconsistency that can be resolved.

To elaborate, we show below the code snippet of the _calculatePeriodFee function defined in both contracts. It comes to our attention that the divisor used in the PriceCalculator contract to calculate the period fee is defined as 1e18 (i.e., IVL_DECIMALS) while the divisor used in the PriceCalculatorUtilization contract to calculate the period fee is defined as 1e16 (i.e., PRICE_DECIMALS * PRICE_MODIFIER_DECIMALS).

```
    /**
     * @notice Calculates and prices in the time value of the option
     * @param amount Option size
     * @param period The option period in seconds (1 days <= period <= 90 days)
     * @return fee The premium size to be paid
     **/
    function _calculatePeriodFee(uint256 amount, uint256 period)
        internal
        view
        virtual
        returns (uint256 fee)
    {
```

```
        return
            (amount * impliedVolRate * period.sqrt()) /
            // priceDecimals /
            IVL_DECIMALS;
    }
```

Listing 3.1: `PriceCalculator::_calculatePeriodFee()`

```
    /**
     * @notice Calculates and prices in the time value of the option
     * @param amount Option size
     * @param period The option period in seconds (1 days <= period <= 90 days)
     * @return fee The premium size to be paid
     **/
    function _calculatePeriodFee(uint256 amount, uint256 period)
        internal
        view
        virtual
        returns (uint256 fee)
    {
        return
            (amount * _priceModifier(amount, period, pool)) /
            PRICE_DECIMALS /
            PRICE_MODIFIER_DECIMALS;
    }
```

Listing 3.2: `PriceCalculatorUtilization::_calculatePeriodFee()`

Recommendation Ensure that the divisors used in both contracts to calculate the period fee are consistent.

Status This issue has been resolved as the team confirms that they are different contracts and the `PriceCalculator` contract is only used for experimental purpose.

3.2 Non-Functional Logic Of HegicStakeAndCover::provide()

- ID: PVE-002
- Severity: Low
- Likelihood: High
- Impact: N/A
- Target: `HegicStakeAndCover`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The `HegicStakeAndCover` contract provides an external `provide()` function for users to deposit tokens into the contract. Users need to deposit `hegicToken` and `baseToken` at the same time. The amount of `hegicToken` to be deposited is provided by the user and the amount of `baseToken` to be deposited relies on the amount of `hegicToken` to be deposited. While examining the routine, we notice the current implementation logic may not work as expected.

To elaborate, we show below its code snippet. Specifically, the execution of `(amount * baseToken.balanceOf(address(this)))/ totalBalance` will always revert (line 186) since the initial value of `totalBalance` is equal to 0.

```
183     /**
184      * @notice Used for depositing tokens into the contract
185      * @param amount The amount of tokens
186      **/
187     function provide(uint256 amount) external {
188         if (profitOf(msg.sender) > 0) claimProfit();
189         uint256 liquidityShare =
190             (amount * baseToken.balanceOf(address(this))) / totalBalance;
191         balanceOf[msg.sender] += amount;
192         startBalance[msg.sender] = shareOf(msg.sender);
193         totalBalance += amount;
194         hegicToken.transferFrom(msg.sender, address(this), amount);
195         baseToken.transferFrom(msg.sender, address(this), liquidityShare);
196         emit Provided(msg.sender, amount, liquidityShare);
197     }
```

Listing 3.3: `HegicStakeAndCover::provide()`

Note a number of routines in the same contract can be similarly improved, including `shareOf()`, `profitOf()`, and `_withdraw()`.

Recommendation Take into consideration the scenario where the initial value of `totalBalance` is equal to 0.

Status This issue has been fixed in the following commit: `2da5c7d`.

3.3 Arithmetic Underflow Avoidance In HegicOperationalTreasury::_replenish()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HegicOperationalTreasury`
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [2]

Description

The `HegicOperationalTreasury` contract provides a privileged function (i.e., `replenish()`) for the admin (with the `DEFAULT_ADMIN_ROLE`) to replenish `baseToken` for the `HegicOperationalTreasury` contract by sending the required amount of `baseToken` from the `HegicStakeAndCover` contract to the `HegicOperationalTreasury` contract. While examining the routine, we notice the current implementation logic can be improved.

To elaborate, we show below the code snippet of the `replenish()/_replenish()` functions. Specifically, if the value of `benchmark` is less than `totalBalance`, the execution of `benchmark + additionalAmount - totalBalance + lockedPremium` will revert (line 181). We point out that if there is a sequence of addition and subtraction operations, it is always better to calculate the addition before the subtraction (on the condition without introducing any extra overflows).

```
    /**
     * @notice Used for replenishing of
     * the Hegic Operational Treasury contract
     **/
    function replenish() external onlyRole(DEFAULT_ADMIN_ROLE) {
        _replenish(0);
    }
```

Listing 3.4: `HegicOperationalTreasury::replenish()`

```
179     function _replenish(uint256 additionalAmount) internal {
180         uint256 transferAmount =
181             benchmark + additionalAmount - totalBalance + lockedPremium;
182         stakeandcoverPool.payOut(transferAmount);
183         totalBalance += transferAmount;
184         emit Replenished(transferAmount);
185     }
```

Listing 3.5: `HegicOperationalTreasury::_replenish()`

Recommendation Revise the above calculation to better mitigate possible execution revert.

Status This issue has been fixed in the following commit: `2da5c7d`.

3.4 Incorrect lockedPremium Update In HegicOperationalTreasury::lockLiquidityFor()

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: HegicOperationalTreasury
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The HegicOperationalTreasury contract provides an external lockLiquidityFor() function for privileged STRATEGY_ROLE to lock liquidity for an active option strategy. Our analysis with this routine shows its current implementation is not correct.

To elaborate, we show below its code snippet. It comes to our attention that the state variable lockedPremium is not updated in the correct order. Specifically, the update for the state variable lockedPremium should precede the calculation of variable availableBalance (lines 91-92).

```
78     /**
79      * @notice Used for locking liquidity in an active options strategy
80      * @param holder The option strategy holder address
81      * @param amount The amount of options strategy contract
82      * @param expiration The options strategy expiration time
83      **/
84     function lockLiquidityFor(
85         address holder,
86         uint128 amount,
87         uint32 expiration
88     ) external override onlyRole(STRATEGY_ROLE) returns (uint256 optionID) {
89         totalLocked += amount;
90         uint128 premium = uint128(_addTokens());
91         uint256 availableBalance =
92             totalBalance + stakeandcoverPool.availableBalance() - lockedPremium;
93         require(totalLocked <= availableBalance, "The amount is too large");
94         require(
95             block.timestamp + maxLockupPeriod >= expiration,
96             "The period is too long"
97         );
98         lockedPremium += premium;
99         lockedByStrategy[msg.sender] += amount;
100        optionID = manager.createOptionFor(holder);
101        lockedLiquidity[optionID] = LockedLiquidity(
102            LockedLiquidityState.Locked,
103            msg.sender,
104            amount,
105            premium,
106            expiration
```

```
        );
    }
```

Listing 3.6: `HegicOperationalTreasury::lockLiquidityFor()`

Recommendation Timely update the state variable `lockedPremium`.

Status This issue has been fixed in the following commit: `2da5c7d`.

3.5 Incorrect withdraw Logic In HegicOperationalTreasury

- ID: PVE-005
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `HegicOperationalTreasury`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The `HegicOperationalTreasury` contract provides an external function (i.e., `withdraw()`) for privileged `DEFAULT_ADMIN_ROLE` to withdraw deposited tokens from the contract. Our analysis with this routine shows its current implementation is not correct.

To elaborate, we show below the code snippet of the `withdraw()/_withdraw()` functions. Its logic is rather straightforward in deducting the withdrawn amount from the internal record and transfer the tokens to the withdrawer. However, the imposed requirement is not correct. Specifically, the requirement `require(amount + totalLocked <= totalBalance)` should be revised as `require(amount + totalLocked + lockedPremium <= taotalBalance + stakeandcoverPool.availableBalance())`, so that the contract keeps a guaranteed amount of tokens for the `Hegic` protocol users.

```
    /**
     * @notice Used for withdrawing deposited
     * tokens from the contract
     * @param to The recipient address
     * @param amount The amount to withdraw
     **/
    function withdraw(address to, uint256 amount)
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
    {
        _withdraw(to, amount);
    }
```

Listing 3.7: `HegicOperationalTreasury::withdraw()`

```
    function _withdraw(address to, uint256 amount) private {
        require(amount + totalLocked <= totalBalance);
        totalBalance -= amount;
        token.transfer(to, amount);
    }
```

Listing 3.8: HegicOperationalTreasury::_withdraw()

Recommendation Revise the require statement to make sure the contract keeps a guaranteed amount of tokens for the Hegic protocol users after the withdraw operation.

Status This issue has been fixed in the following commit: 83aa7e1.

3.6 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple contracts
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

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 3.9: 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 current implementation, if we examine the `HegicOperationalTreasury::_withdraw()` routine that is designed to withdraw `token` from the contract. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 208).

```
205     function _withdraw(address to, uint256 amount) private {
206         require(amount + totalLocked <= totalBalance);
207         totalBalance -= amount;
208         token.transfer(to, amount);
209     }
```

Listing 3.10: `HegicOperationalTreasury::_withdraw()`

Note this issue is also applicable to other routines, including `transfer()/payOut()/_withdraw()/providd()` from the `HegicStakeAndCover` contract.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()/transferFrom()`.

Status This issue has been fixed in the following commit: `83aa7e1`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the Hegic protocol, there are five privileged accounts, i.e., owner, DEFAULT_ADMIN_ROLE, HEGIC_POOL_ROLE , STRATEGY_ROLE, and SELLER_ROLE. These accounts play a critical role in governing and regulating the protocol-wide operations (e.g., withdraw tokens from the HegicOperationalTreasury contract, replenish tokens for the HegicOperationalTreasury contract, withdraw the deposited tokens from the HegicStakeAndCover contract, disable withdrawing tokens from the HegicStakeAndCover contract, create option for a specified account, lock liquidity, sell option, and set the key parameters, etc.).

In the following, we use the HegicOperationalTreasury contract as an example and show the representative functions potentially affected by the privileges of the DEFAULT_ADMIN_ROLE/STRATEGY_ROLE accounts.

```
    /**
     * @notice Used for withdrawing deposited
     * tokens from the contract
     * @param to The recipient address
     * @param amount The amount to withdraw
     **/
    function withdraw(address to, uint256 amount)
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
    {
        _withdraw(to, amount);
    }

    /**
     * @notice Used for replenishing of
     * the Hegic Operational Treasury contract
     **/
    function replenish() external onlyRole(DEFAULT_ADMIN_ROLE) {
        _replenish(0);
    }

    /**
     * @notice Used for locking liquidity in an active options strategy
     * @param holder The option strategy holder address
     * @param amount The amount of options strategy contract
     * @param expiration The options strategy expiration time
```

```
     **/
    function lockLiquidityFor(
        address holder,
        uint128 amount,
        uint32 expiration
    ) external override onlyRole(STRATEGY_ROLE) returns (uint256 optionID) {
        totalLocked += amount;
        uint128 premium = uint128(_addTokens());
        uint256 availableBalance =
            totalBalance + stakeandcoverPool.availableBalance() - lockedPremium;
        require(totalLocked <= availableBalance, "The amount is too large");
        require(
            block.timestamp + maxLockupPeriod >= expiration,
            "The period is too long"
        );
        lockedPremium += premium;
        lockedByStrategy[msg.sender] += amount;
        optionID = manager.createOptionFor(holder);
        lockedLiquidity[optionID] = LockedLiquidity(
            LockedLiquidityState.Locked,
            msg.sender,
            amount,
            premium,
            expiration
        );
    }

    /**
     * @notice Used for setting the initial
     * contract benchmark for calculating
     * future profits or losses
     * @param value The benchmark value
     **/
    function setBenchmark(uint256 value) external onlyRole(DEFAULT_ADMIN_ROLE) {
        benchmark = value;
    }
```

Listing 3.11: `HegicOperationalTreasury::withdraw()/replenish()/lockLiquidityFor()/setBenchmark()`

```
    /**
     * @notice Used for adding deposited tokens
     * (e.g. premiums) to the contract's totalBalance
     * @param amount The amount of tokens to add
     **/
    function addTokens()
        public
        onlyRole(DEFAULT_ADMIN_ROLE)
        returns (uint256 amount)
    {
        return _addTokens();
    }
```

Listing 3.12: `HegicOperationalTreasury::addTokens()`

The first function `withdraw()` allows for the `DEFAULT_ADMIN_ROLE` to withdraw tokens from the `HegicOperationalTreasury` contract. The second function `replenish()` allows for the `DEFAULT_ADMIN_ROLE` to replenish `baseToken` for the `HegicOperationalTreasury` contract. The third function `lockLiquidityFor()` allows for the `STRATEGY_ROLE` to lock liquidity for an active option strategy. The fourth function `setBenchmark()` allows for the `DEFAULT_ADMIN_ROLE` to set the `benchmark` for the `HegicOperationalTreasury` contract. And the fifth function function `addTokens()` allows for the `DEFAULT_ADMIN_ROLE` to add deposited tokens (e.g. premiums) to the contract's totalBalance. We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to privileged accounts explicit to `Hegic` protocol users

Status This issue has been confirmed.

3.8 Improved Sanity Checks For System/Function Parameters

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HegicPool`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Hegic` protocol is no exception. Specifically, if we examine the `HegicPool` contract, it has defined a number of protocol-wide risk parameters, such as `maxDepositAmount`, `collateralizationRatio`, and `pricer`. In the following, we show the corresponding routines that allow for their changes.

```
    /**
     * @notice Used for setting the total maximum amount
     * that could be deposited into the pools contracts.
     * Note that different total maximum amounts could be set
     * for the hedged and unhedged - classic - liquidity tranches.
     * @param total Maximum amount of assets in the pool
     * in hedged and unhedged (classic) liquidity tranches combined
     **/
    function setMaxDepositAmount(uint256 total)
```

```
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
    {
        maxDepositAmount = total;
    }
```

Listing 3.13: `HegicPool::setMaxDepositAmount()`

```
    /**
     * @notice Used for setting the collateralization ratio for the option
     * collateral size that will be locked at the moment of buying them.
     *
     * Example: if `CollateralizationRatio` = 50, then 50% of an option's
     * notional size will be locked in the pools at the moment of buying it:
     * say, 1 ETH call option will be collateralized with 0.5 ETH (50%).
     * Note that if an option holder's net P&L USD value (as options
     * are cash-settled) will exceed the amount of the collateral locked
     * in the option, she will receive the required amount at the moment
     * of exercising the option using the pool's unutilized (unlocked) funds.
     * @param value The collateralization ratio in a range of 30% - 50%
     **/
    function setCollateralizationRatio(uint256 value)
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
    {
        collateralizationRatio = value;
    }
```

Listing 3.14: `HegicPool::setCollateralizationRatio()`

```
    /**
     * @notice Used for setting the price calculator
     * contract that will be used for pricing the options.
     * @param pc A new price calculator contract address
     **/
    function setPriceCalculator(IPriceCalculator pc)
        public
        onlyRole(DEFAULT_ADMIN_ROLE)
    {
        pricer = pc;
    }
```

Listing 3.15: `HegicPool::setPriceCalculator()`

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these 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 `collateralizationRatio` may lock an unreasonable amount of option collateral at the moment of buying the option.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status This issue has been fixed in the following commit: `2da5c7d`.

3.9 Improved Reentrancy Protection In HegicPool

- ID: PVE-009
- Severity: Undetermined
- Likelihood: Low
- Impact: Low
- Target: `HegicPool`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [4]

Description

In the `HegicPool` contract, we notice the `exercise()` function is used to exercise the `ITM` (in-the-money) option contract in case of having the unrealized profits accrued during the period of holding the option contract. Our analysis shows there is a potential reentrancy issue in the function.

To elaborate, we show below the code snippet of the `exercise()` function. In the function, the `_send()` function will be called (line 266) to transfer `token` from the pool to the owner of the option. If the `token` faithfully implements the ERC777-like standard, then the `exercise()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend()` and `tokensReceived()` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in `token.safeTransfer(to, transferAmount)` (line 274) before the actual transfer of the underlying assets occurs. So far, we also do not know how an attacker can exploit this issue to earn profit. After internal discussion, we consider it is necessary to bring this issue up to the team. Though the implementation of the `exercise()` function is well designed and meets the `Checks-Effects-Interactions` pattern, we may intend to use the `ReentrancyGuard::nonReentrant` modifier to protect the `exercise()` and `sellOption()` functions at the whole protocol level.

```
243     /**
```

```
     * @notice Used for exercising the ITM (in-the-money)
     * options contracts in case of having the unrealized profits
     * accrued during the period of holding the option contract.
     * @param id ID of ERC721 token linked to the option
     **/
    function exercise(uint256 id) external override {
        Option storage option = options[id];
        uint256 profit = _profitOf(option);
        require(
            optionsManager.isApprovedOrOwner(_msgSender(), id),
            "Pool Error: msg.sender can't exercise this option"
        );
        require(
            option.expired > block.timestamp,
            "Pool Error: The option has already expired"
        );
        require(
            profit > 0,
            "Pool Error: There are no unrealized profits for this option"
        );
        _unlock(option);
        option.state = OptionState.Exercised;
        _send(optionsManager.ownerOf(id), profit);
        emit Exercised(id, profit);
    }

    function _send(address to, uint256 transferAmount) private {
        require(to != address(0));

        totalBalance -= transferAmount;
        token.safeTransfer(to, transferAmount);
    }
```

Listing 3.16: `HegicPool::exercise()/_send()`

Recommendation Apply the non-reentrancy protection in all above-mentioned routines.

Status This issue has been fixed in the following commit: `9387bbc`.

4 | Conclusion

In this audit, we have analyzed the `Hegic` design and implementation. The `Hegic` protocol is an on-chain peer-to-pool options trading protocol built on `Ethereum`. The current code base is well organized and those identified issues are promptly confirmed and addressed.

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-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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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