SMART CONTRACT AUDIT REPORT

for

Hegic Protocol (v8888)

Prepared By: Yiqun Chen

PeckShield

August 10, 2021

Document Properties

Client	Hegic
Title	Smart Contract Audit Report
Target	Hegic
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 10, 2021	Shulin Bie	Final Release
1.0-rc	August 8, 2021	Shulin Bie	Release Candidate

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

Given the opportunity to review the design document and related smart contract source code of the Hegic protocol (v8888), 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

Hegic was founded a year and a half ago in February, 2020. The Hegic protocol is an on-chain peer-to-pool options trading protocol built on Ethereum. With the Hegic protocol, DeFi and crypto users can trade 24/7, cash-settled, on-chain ETH and WBTC call/put options with no KYC or registration required for trading. 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

Item	Description
Target	Hegic
Website	https://www.hegic.co/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 10, 2021

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

- https://github.com/hegic/Hegic-protocol-v8888.git (2dcd44d)

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-protocol-v8888.git (a851533)

1.2 About PeckShield

PeckShield Inc. [14] 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 [13]:

- 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) [12], 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 `Hegic` (v8888) 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 logic, 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	1	
Low	3	
Informational	0	
Undetermined	1	
Total	6	

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 1 high-severity vulnerability, 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Hegic Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improper Logic In PriceCalculator::_priceModifier()	Coding Practices	Fixed
PVE-002	High	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-003	Low	Potential Overflow In HegicMath::sqrt()	Numeric Errors	Fixed
PVE-004	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-005	Medium	Potential Sandwich/MEV Attack For createOption()	Time and State	Fixed
PVE-006	Undetermined	Revisited Reentrancy Protection In HegicPool	Time and State	Fixed

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

3 | Detailed Results

3.1 Improper Logic In PriceCalculator::_priceModifier()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `PriceCalculator`
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [5]

Description

In the `Hegic` protocol, the `PriceCalculator` contract is designed to calculate the `premium` and `settlementFee` of the options. While examining the various functionalities in the `PriceCalculator` contract, we notice the logic of the `_priceModifier()` function needs to be revised.

To elaborate, we show below the related code snippet of the contract. In the `_calcualtePeriodFee` function that is used to calculate the total fee of the option(including the `premium` and `settlementFee`), it has the `_priceModifier()` that is used (line 99) to calculate the fee of the option per unit of underlying asset. It comes to our attention that the public `utilizationRate` storage variable will always be 0 because there is no function to update it in the contract. From this, we believe the statements from line 118 to line 123 will not take effect and suggest to remove them safely (or add the function to update the `utilizationRate` variable).

```
87     /**
88      * @notice Calculates and prices in the time value of the option
89      * @param amount Option size
90      * @param period The option period in seconds (1 days <= period <= 90 days)
91      * @return fee The premium size to be paid
92      **/
93     function _calcualtePeriodFee(uint256 amount, uint256 period)
94         internal
95         view
96         returns (uint256 fee)
97     {
98         return
```

```
            (amount * _priceModifier(amount, period, pool)) /
            PRICE_DECIMALS /
            PRICE_MODIFIER_DECIMALS;
    }

    /**
     * @notice Calculates 'periodFee' of the option
     * @param amount The option size
     * @param period The option period in seconds (1 days <= period <= 90 days)
     **/
    function _priceModifier(
        uint256 amount,
        uint256 period,
        IHegicPool pool
    ) internal view returns (uint256 iv) {
        uint256 poolBalance = pool.totalBalance();
        require(poolBalance > 0, "Pool Error: The pool is empty");
        iv = impliedVolRate * period.sqrt();

        uint256 lockedAmount = pool.lockedAmount() + amount;
        uint256 utilization = (lockedAmount * 100e8) / poolBalance;

        if (utilization > 40e8) {
            iv += (iv * (utilization - 40e8) * utilizationRate) / 40e16;
        }
    }
```

Listing 3.1: `PriceCalculator::_calcualtePeriodFee()&&_priceModifier()`

Recommendation Revisit the logic of the `_priceModifier()` function or add the function to update the `utilizationRate` variable.

Status The issue has been addressed by the following commit: `f0b6606`.

3.2 Trust Issue Of Admin Keys

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `Hegic` protocol, there is a privileged account (with the `DEFAULT_ADMIN_ROLE`) that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring system parameters and performing privileged operations). Especially, the privileged account has the ability to transfer all the assets out of the Hegic Pool.

To elaborate, we show below the related code snippet of the contract. The settlementFeeRecipient account that is assigned by the privileged account will get the approval of the Hegic Pool after calling the approve() function. And then the settlementFeeRecipient account will have the ability to transfer all the assets out of the Hegic Pool.

```
    /**
     * @notice Used for approving the staking contracts
     * to receive the `settlementFee` in ERC20 tokens
     * that will be accumulated and distributed in
     * staking rewards among the staking participants.
     **/
    function approve() public {
        token.approve(address(settlementFeeRecipient), type(uint256).max);
    }
```

Listing 3.2: HegicPool::approve()

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the Hegic design. Especially, it is not safe to assign the approval of the Hegic Pool to the settlementFeeRecipient account.

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. Additionally, it is not necessary to assign the approval of the Hegic Pool to the settlementFeeRecipient account.

Status The issue has been mitigated by the following commit: b8af917.

3.3 Potential Overflow In HegicMath::sqrt()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HegicMath`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [2]

Description

In the `Hegic` protocol, the `HegicMath` library is designed as math utilities missing in the Solidity language. While examining the library, we notice there is a potential overflow in the `sqrt()` function.

To elaborate, we show below the code snippet of the `sqrt()` function. The `sqrt()` function is used to calculate a square root of the number. The input parameter type of the `sqrt()` is `uint256` whose value range is from 0 to $2 ** 256 - 1$. We notice the `uint256 k = (x + 1)>> 1` (line 29) will overflow if the input parameter is $2 ** 256 - 1$. We may intend to replace it with the `uint256 k = (x >> 1)+ 1`.

```
23     /**
24      * @dev Calculates a square root of the number.
25      * Responds with an "invalid opcode" at uint(-1).
26      **/
27     function sqrt(uint256 x) internal pure returns (uint256 result) {
28         result = x;
29         uint256 k = (x + 1) >> 1;
30         while (k < result) (result, k) = (k, (x / k + k) >> 1);
31     }
```

Listing 3.3: `HegicMath::sqrt()`

Recommendation We show below the improved implementation of the `sqrt()` function.

```
23     /**
24      * @dev Calculates a square root of the number.
25      * Responds with an "invalid opcode" at uint(-1).
26      **/
27     function sqrt(uint256 x) internal pure returns (uint256 result) {
28         result = x;
29         uint256 k = (x >> 1) + 1;
30         while (k < result) (result, k) = (k, (x / k + k) >> 1);
31     }
```

Listing 3.4: `HegicMath::sqrt()`

Status The issue has been addressed by the following commit: `11a37eb`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Facade/HegicPool`
- Category: Coding Practices [9]
- 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 this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Listing 3.5: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `approve()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount)external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of `USDT`'s `approve()`, the call will be unfortunately reverted.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, 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.

In the following, we show the `poolApprove()` routine in the `Facade` contract. If the `USDT` token is supported as `pool.token()`, the unsafe version of `pool.token().approve(address(pool), type(uint256).max)` (line 123) may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```
119     /**
120      * @notice Used for approving the pools contracts addresses.
121      **/
122     function poolApprove(IHegicPool pool) external {
123         pool.token().approve(address(pool), type(uint256).max);
124     }
```

Listing 3.6: `Facade::poolApprove()`

Note the `Facade::createOption()` and `HegicPool::approve()` routines can be similarly improved.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been addressed by the following commits: `b2dd886`.

3.5 Potential Sandwich/MEV Attack For createOption()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Facade`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [6]

Description

In the `Facade` contract, the `createOption()` function is designed to allow buyers to purchase the put/call options. Our analysis shows there is a potential `Sandwich/MEV` attack for the `createOption()` function.

To elaborate, we show below the related code snippet of the contract. In the `createOption()` function, if the payment token of the buyer is not the underlying token of the `Hegic Pool`, the `swapTokensForExactTokens()` function (line 152) of `UniswapV2` will be called to swap the payment token of the buyer to the underlying token of the `Hegic Pool`. We notice the `optionPrice` is calculated by the `exchange.getAmountsIn(_baseTotal, swappath)[0]` (line 82), which is the actual amount of the payment token that will be swapped. However, it is assigned to the `amountInMax` of the `swapTokensForExactTokens()` function, which means the restriction on possible slippage will never take effect and is therefore vulnerable to possible front-running attacks.

```
134     function createOption(
135         IHegicPool pool,
136         uint256 period,
137         uint256 amount,
138         uint256 strike,
139         address[] calldata swappath
140     ) external payable {
141         address buyer = _msgSender();
142         (uint256 optionPrice, uint256 rawOptionPrice, , ) =
143             getOptionPrice(pool, period, amount, strike, swappath);
144         IERC20 paymentToken = IERC20(swappath[0]);
145         paymentToken.safeTransferFrom(buyer, address(this), optionPrice);
146         if (swappath.length > 1) {
147             if (
148                 paymentToken.allowance(address(this), address(exchange)) <
149                 optionPrice
150             ) paymentToken.approve(address(exchange), type(uint256).max);

152             exchange.swapTokensForExactTokens(
153                 rawOptionPrice,
154                 optionPrice,
155                 swappath,
156                 address(this),
157                 block.timestamp
158             );
159         }
160         pool.sellOption(buyer, period, amount, strike);
161     }
```

Listing 3.7: `Facade::createOption()`

```
63      function getOptionPrice(
64          IHegicPool pool,
65          uint256 period,
66          uint256 amount,
67          uint256 strike,
68          address[] calldata swappath
69      )
70          public
71          view
72          returns (
```

```
            uint256 total,
            uint256 baseTotal,
            uint256 settlementFee,
            uint256 premium
        )
    {
        (uint256 _baseTotal, uint256 baseSettlementFee, uint256 basePremium) =
            getBaseOptionCost(pool, period, amount, strike);
        if (swappath.length > 1)
            total = exchange.getAmountsIn(_baseTotal, swappath)[0];
        else total = _baseTotal;

        baseTotal = _baseTotal;
        settlementFee = (total * baseSettlementFee) / baseTotal;
        premium = (total * basePremium) / baseTotal;
    }
```

Listing 3.8: `Facade::getOptionPrice()`

Recommendation Improve the `createOption()` function by adding necessary slippage control.

Status The issue has been addressed by the following commit: `d159a97`.

3.6 Revisited Reentrancy Protection In HegicPool

- ID: PVE-006
- Severity: Undetermined
- Likelihood: Low
- Impact: Low
- Target: `HegicPool`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [4]

Description

In the `HegicPool` contract, we notice the `provideFrom()` function is used to deposit the funds into the `Hegic Pool` and mint the ERC721 token, which represents the liquidity provider's share in the `Hegic Pool`. Our analysis shows there is a potential reentrancy vulnerability in the function.

To elaborate, we show below the code snippet of the `provideFrom()` function. In the function, the `_safeMint()` function will be called (line 406) to mint an ERC721 token for the liquidity provider. A further examination of `_safeMint()` of ERC721 shows the `_checkOnERC721Received()` function will be called to ensure the recipient confirms the receipt. If the recipient is an evil attacker, she may launch a re-entrancy attack in the callback function. So far, we also do not know how an attacker can exploit this vulnerability to earn profit. After internal discussion, we consider it is necessary to bring this vulnerability up to the team. Though the implementation of the `provideFrom()` function is well designed and meets the `Checks-Effects-Interactions` pattern, we may intend to use the `ReentrancyGuard::nonReentrant` modifier to protect the `provideFrom()`, `withdraw()` and `withdrawWithoutHedge()` functions at the whole protocol level.

```
    function provideFrom(
        address account,
        uint256 amount,
        bool hedged,
        uint256 minShare
    ) external override returns (uint256 share) {
        uint256 totalShare = hedged ? hedgedShare : unhedgedShare;
        uint256 balance = hedged ? hedgedBalance : unhedgedBalance;
        share = totalShare > 0 && balance > 0
            ? (amount * totalShare) / balance
            : amount * INITIAL_RATE;
        uint256 limit =
            hedged
                ? maxHedgedDepositAmount - hedgedBalance
                : maxDepositAmount - hedgedBalance - unhedgedBalance;
        require(share >= minShare, "Pool Error: The mint limit is too large");
        require(share > 0, "Pool Error: The amount is too small");
        require(
            amount <= limit,
            "Pool Error: Depositing into the pool is not available"
        );

        if (hedged) {
            hedgedShare += share;
            hedgedBalance += amount;
        } else {
            unhedgedShare += share;
            unhedgedBalance += amount;
        }

        uint256 trancheID = tranches.length;
        tranches.push(
            Tranche(TrancheState.Open, share, amount, block.timestamp, hedged)
        );
        _safeMint(account, trancheID);
        token.safeTransferFrom(_msgSender(), address(this), amount);
    }
```

Listing 3.9: `HegicPool::provideFrom()`

```
    function _safeMint(
        address to,
        uint256 tokenId,
        bytes memory _data
    ) internal virtual {
        _mint(to, tokenId);
        require(
            _checkOnERC721Received(address(0), to, tokenId, _data),
```

```
            "ERC721: transfer to non ERC721Receiver implementer"
        );
    }

    ...

    function _checkOnERC721Received(
        address from,
        address to,
        uint256 tokenId,
        bytes memory _data
    ) private returns (bool) {
        if (to.isContract()) {
            try IERC721Receiver(to).onERC721Received(_msgSender(), from, tokenId, _data)
                returns (bytes4 retval) {
                return retval == IERC721Receiver(to).onERC721Received.selector;
            } catch (bytes memory reason) {
                if (reason.length == 0) {
                    revert("ERC721: transfer to non ERC721Receiver implementer");
                } else {
                    assembly {
                        revert(add(32, reason), mload(reason))
                    }
                }
            }
        } else {
            return true;
        }
    }
```

Listing 3.10: `ERC721::_safeMint()&&_checkOnERC721Received()`

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

Status The issue has been addressed by the following commits: `5f70a47` && `925f94e`.

4 Conclusion

In this audit, we have analyzed the `Hegic` (v8888) design and implementation. The `Hegic` protocol is an on-chain peer-to-pool options trading protocol built on Ethereum. With the `Hegic` protocol, DeFi and crypto users can trade 24/7, cash-settled, on-chain ETH and WBTC call/put options with no KYC or registration required for trading. 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 current code base is well organized and those identified issues are promptly confirmed and fixed.

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

References

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

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

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

[8] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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

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