SMART CONTRACT AUDIT REPORT

for

HEGIC

Prepared By: Shuxiao Wang

Hangzhou, China
October 1, 2020

Document Properties

Client	Hegic
Title	Smart Contract Audit Report
Target	Hegic
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Jeff Liu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 1, 2020	Xuxian Jiang	Final Release
1.0-rc2	September 30, 2020	Xuxian Jiang	Release Candidate #2
1.0-rc1	September 28, 2020	Xuxian Jiang	Release Candidate #1
0.4	September 15, 2020	Xuxian Jiang	Additional Findings #3
0.3	September 10, 2020	Xuxian Jiang	Additional Findings #2
0.2	September 8, 2020	Xuxian Jiang	Additional Findings #1
0.1	September 4, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the **Hegic Protocol** design document and related smart contract source code, we in the report outline 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 Protocol

Hegic is a protocol for trustless creation, maintenance, and settlement of hedge contracts. A hedge contract is an options-like on-chain contract that gives the holder (buyer) a right to buy or to sell an asset at a certain price as well as imposes the obligation on the writer (seller) to buy or to sell an asset during a certain time period. The Hegic Protocol plays the role of the Options Clearing Corporation (OCC) in traditional financial markets, but in a trustless, non-custodial manner. It can be useful for participants who want to protect their assets from the price downside and for the liquidity providers who might find the returns on writing hedge contracts attractive. Hegic 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
Issuer	Hegic
Website	https://www.hegic.co/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 1, 2020

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

- https://github.com/hegic/contracts (2943e24)

1.2 About PeckShield

PeckShield Inc. [15] 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 [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the Hegic Protocol design and implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	1	■
High	0	
Medium	4	■ ■ ■ ■
Low	2	■ ■
Informational	5	■ ■ ■ ■ ■
Total	12	

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 critical-severity vulnerability, 4 medium-severity vulnerabilities, 2 low-severity vulnerabilities and 5 informational recommendations.

Table 2.1: Key Hegic Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Non-Functional Lockup Periods in HegicStaking	Business Logics	Fixed
PVE-002	Low	Possible Front-Running Against Pool Withdrawals And Staking	Business Logics	Partially Fixed
PVE-003	Medium	Bypass of Daily Reward Limit in HegicRewards	Business Logics	Fixed
PVE-004	Informational	Improved Precision With Division Avoidance	Coding Practices	Fixed
PVE-005	Informational	Improved Precision With Ceiling Division	Numeric Errors	Fixed
PVE-006	Informational	Less Friction For Option Creation	Coding Practices	Fixed
PVE-007	Medium	Wrong Reward Rate in HegicWBTCRewards	Business Logics	Fixed
PVE-008	Informational	Suggested Reservation of The First enum Element	Coding Practices	Fixed
PVE-009	Low	Enhanced Business Logic of lock() in HegicETHPool	Business Logics	Fixed
PVE-010	Informational	Redundant lockupFree Verification	Business Logics	Fixed
PVE-011	Medium	Denial-of-Service in getReward()	Business Logics	Fixed
PVE-012	Critical	Option Pool Draining With Invalid optionType	Business Logics	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 Non-Functional Lockup Periods in HegicStaking

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `HegicStakingETH/HegicStakingWBTC`
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

By design, the Hegic protocol will generate and collect settlement fees (in `ETH` and `WBTC`) paid every time when a hegic option contract is purchased. The `HEGIC` token holders can stake their tokens to receive pro-rata staking rewards. For example, if there will be 10 active staking lots, each of them will be receiving 10% of rewards; and if there will be 100 active staking lots, each of them will be receiving 1% of rewards.

The staking logic is implemented in two contracts: `HegicStakingETH` and `HegicStakingWBTC`. As the names indicate, they are pool-specific. In order to prevent possible flashloan-assisted front-running attacks that may claim the majority of rewards, the staking logic is designed to have a lockup period for staked assets. For each account, the associated lockup period is recorded as `[lastBoughtTimestamp[account], lastBoughtTimestamp[account].add(lockupPeriod)]`.

```
    function buy(uint amount) external override {
        require(amount > 0, "Amount is zero");
        require(totalSupply() + amount <= MAX_SUPPLY);
        _mint(msg.sender, amount);
        HEGIC.safeTransferFrom(msg.sender, address(this), amount.mul(LOT_PRICE));
    }
```

Listing 3.1: HegicStaking.sol

However, it comes to our attention that the staking function, i.e., `buy()`, does not record the `lastBoughtTimestamp` of the buyer. As a result, the protocol keeps the default `lastBoughtTimestamp` of 0

for the buyer. When any `transfer()` or `transferFrom()` action occurs, the lockup verification routine, i.e., `_beforeTokenTransfer()`, always gives a green light even when the transferred staking pool tokens are just bought! In other words, despite the fact the receiver may have explicitly requested for not accepting any funds in the lockup period: `_revertTransfersInLockUpPeriod[receiver] == true` (line 108), the transfer will not be blocked.

```
100     function _beforeTokenTransfer(address from, address to, uint256) internal override {
101         if (from != address(0)) saveProfit(from);
102         if (to != address(0)) saveProfit(to);
103         if (
104             lastBoughtTimestamp[from].add(lockupPeriod) > block.timestamp &&
105             lastBoughtTimestamp[from] > lastBoughtTimestamp[to]
106         ) {
107             require(
108                 !_revertTransfersInLockUpPeriod[to],
109                 "the recipient does not accept blocked funds"
110             );
111             lastBoughtTimestamp[to] = lastBoughtTimestamp[from];
112         }
113     }
```

Listing 3.2: HegicStaking.sol

Similarly, since the protocol always keeps the default `_beforeTokenTransfer()` value for any account, the `lockupFree` modifier () `require(lastBoughtTimestamp[msg.sender].add(lockupPeriod)<= block.timestamp)`) is always satisfied, meaning any stakers can immediately sell without being locked.

```
    function sell(uint amount) external override lockupFree {
        _burn(msg.sender, amount);
        HEGIC.safeTransfer(msg.sender, amount.mul(LOT_PRICE));
    }
```

Listing 3.3: HegicStaking.sol

Recommendation Properly record the `lastBoughtTimestamp` when a `HEGIC` holder stakes the assets as follows:

```
    function buy(uint amount) external override {
        require(amount > 0, "Amount is zero");
        require(totalSupply() + amount <= MAX_SUPPLY);
        lastBoughtTimestamp[msg.sender] = block.timestamp;
        _mint(msg.sender, amount);
        HEGIC.safeTransferFrom(msg.sender, address(this), amount.mul(LOT_PRICE));
    }
```

Listing 3.4: HegicStaking.sol

Status This issue has been fixed in the commit: 83499168bbbf622cae53527e49576e340d06be8c.

3.2 Possible Front-Running Against Pool Withdrawals And Staking

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: HegicETHPool/HegicERCPool
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

Hegic is an on-chain peer-to-pool options trading protocol built on Ethereum. The pool has well-defined APIs that allow for liquidity providers ("writers") to efficiently add or remove funds. By doing so, funds from liquidity providers can be distributed among many hedge contracts simultaneously. It not only diversifies the liquidity allocation and makes efficient use of funds in the pool, but collectively shares the associated risks from one particular writer to all active liquidity providers.

The defined APIs for pool management mainly include `provide()` and `withdraw()`. The `provide()` routine is used to add funds into the pool while the `withdraw()` routine is used to withdraw funds from the pool. Similar to the management of staked assets (Section 3.1), the pool supports a lockup period for new funds into the pool. Specifically, for each liquidity provider, the associated lockup period is recorded as `[lastProvideTimestamp[account], lastProvideTimestamp[account].add(lockupPeriod)]`. Moreover, when any `transfer()` or `transferFrom()` action occurs, there is an accompanying lockup verification routine, i.e., `_beforeTokenTransfer()`. In the following, we outline the code logic of the three related functions: `provide()`, `withdraw()`, and `_beforeTokenTransfer()`.

```
    function withdraw(uint256 amount, uint256 maxBurn) external returns (uint256 burn) {
        require(
            lastProvideTimestamp[msg.sender].add(lockupPeriod) <= block.timestamp,
            "Pool: Withdrawal is locked up"
        );
        require(
            amount <= availableBalance(),
            "Pool Error: Not enough funds on the pool contract. Please lower the amount."
        );
        burn = amount.mul(totalSupply()).div(totalBalance());

        require(burn <= maxBurn, "Pool: Burn limit is too small");
        require(burn <= balanceOf(msg.sender), "Pool: Amount is too large");
        require(burn > 0, "Pool: Amount is too small");

        _burn(msg.sender, burn);
        emit Withdraw(msg.sender, amount, burn);
        msg.sender.transfer(amount);
```

```
    }
```

Listing 3.5: HegicETHPool.sol

```
88     function withdraw(uint256 amount, uint256 maxBurn) external returns (uint256 burn) {
89         require(
90             lastProvideTimestamp[msg.sender].add(lockupPeriod) <= block.timestamp,
91             "Pool: Withdrawal is locked up"
92         );
93         require(
94             amount <= availableBalance(),
95             "Pool Error: Not enough funds on the pool contract. Please lower the amount."
96         );
97         burn = amount.mul(totalSupply()).div(totalBalance());
98 
99         require(burn <= maxBurn, "Pool: Burn limit is too small");
100        require(burn <= balanceOf(msg.sender), "Pool: Amount is too large");
101        require(burn > 0, "Pool: Amount is too small");
102
103        _burn(msg.sender, burn);
104        emit Withdraw(msg.sender, amount, burn);
105        msg.sender.transfer(amount);
106    }
```

Listing 3.6: HegicETHPool.sol

```
194    function _beforeTokenTransfer(address from, address to, uint256) internal override {
195        if (
196            lastProvideTimestamp[from].add(lockupPeriod) > block.timestamp &&
197            lastProvideTimestamp[from] > lastProvideTimestamp[to]
198        ) {
199            require(
200                !_revertTransfersInLockUpPeriod[to],
201                "the recipient does not accept blocked funds"
202            );
203            lastProvideTimestamp[to] = lastProvideTimestamp[from];
204        }
205    }
```

Listing 3.7: HegicETHPool.sol

By examining these three routines, we identify a possible front-running attack that may block an ongoing withdrawal attempt. Specifically, when a `transfer()` or `transferFrom()` action occurs, the lockup period of the receiver, i.e.,`lastProvideTimestamp[to]`, might be accordingly updated (line 203). Therefore, upon the observation of a `withdraw()` attempt from a victim, a malicious actor could intentionally transfer 1 `WEI` to the victim. By doing so, the `lastProvideTimestamp` of the victim is updated with the `lastProvideTimestamp` of the malicious actor. As a result, the specific `withdraw()` attempt is blocked as it occurs in the lockup period (line 90). We emphasize this attack will not work for those victims who do turn on the `lastProvideTimestamp` flag. However, most victims likely will not turn the flag on since it requires an extra transaction to achieve that.

In addition, the staking support in Hegic (implemented in `HegicStakingETH` and `HegicStakingWBTC`) shares a similar issue as the $address(0)$ could be contaminated, hence blocking all `buy()` attempts from legitimate stakers who turn on the `_revertTransfersInLockUpPeriod` flag. Note this attack does not work for victims who have not turned the flag on, which is contrary to the pool case.

Last, we observe similar front-running attacks in other settings, such as front-running the `swapToWBTC()` routine of the `HegicWBTCOptions` contract (when a `WBTC` option is being created) or the `send()` routine of the `HegicWBTCPool` contract (when a pool loss is forthcoming).

Recommendation A mitigation to the above front-running attacks need to turn on (the pool front-running) or off (the stake front-running) the victim's flag, i.e., `_revertTransfersInLockUpPeriod`. By doing so, we can prevent the `lastProvideTimestamp` flag from being manipulated by others.

Status This issue has been addressed in the commit: 83499168bbbf622cae53527e49576e340d06be8c. In the meantime, we acknowledge that front-running attacks are inherent in current DEXes and there is still a need to search for more effective countermeasures.

3.3 Bypass of Daily Reward Limit in HegicRewards

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `HegicRewards`
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

Hegic provides an incentivization mechanism to encourage early adoption. Specifically, certain `HEGIC` tokens will be distributed pro-rata on a daily basis among options holders. The logic has been implemented in the `HegicRewards` contract. The current implementation sets the maximum daily reward as `MAX_DAILY_REWARD = 165_000e18`.

Each option holder is eligible to claim the rewards via the `getReward()` routine. The logic is rather straightforward in firstly determining the reward amount, next marking the option's reward-claiming status, then ensuring the rewarded amount still falls within the daily rewarding limit, and finally transferring the reward.

Our analysis shows that the above logic forgets to update the daily rewarded amount that has been claimed so far. Therefore, the above checking of daily rewarding limit translates into that each individual option reward is no more than the daily limit.

```
    function getReward(uint optionId) external {
        uint amount = rewardAmount(optionId);
        uint today = block.timestamp / 1 days;
        (, address holder, , , , , ) = hegicOptions.options(optionId);
        require(!rewardedOptions[optionId], "The option was rewarded");
        require(
            amount.add(dailyReward[today]) < MAX_DAILY_REWARD,
            "Exceeds daily limits"
        );
        rewardedOptions[optionId] = true;
        hegic.safeTransfer(holder, amount);
    }
```

Listing 3.8: HegicRewards.sol

Recommendation Revise the logic to properly implement the daily reward limit. An example revision is shown below:

```
    function getReward(uint optionId) external {
        uint amount = rewardAmount(optionId);
        uint today = block.timestamp / 1 days;
        (, address holder, , , , , ) = hegicOptions.options(optionId);
        require(!rewardedOptions[optionId], "The option was rewarded");
        require(
            amount.add(dailyReward[today]) < MAX_DAILY_REWARD,
            "Exceeds daily limits"
        );
        rewardedOptions[optionId] = true;
        dailyReward[today] = dailyReward[today].add(amount);
        hegic.safeTransfer(holder, amount);
    }
```

Listing 3.9: HegicRewards.sol

Status This issue has been fixed in the commit: 83499168bbbf622cae53527e49576e340d06be8c.

3.4 Improved Precision With Division Avoidance

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: HegicETHPool/HegicERCPool
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.2, Hegic is an on-chain peer-to-pool options trading protocol. For each option being purchased, the pool will lock certain amount of funds to meet the need in case the option will be exercised (at the agreed strike price) within the option's validity period. Internally, a local variable named `lockedAmount` keeps track of total locked amount of funds in current pool for active options.

The Hegic protocol takes a rather prudent approach in maintaining a threshold of 80% of locked funds above which no new option will be created. This restriction is enforced when a new option always needs to lock certain amount of funds in the pool (in the `lock()` routine as shown below), i.e., `require(lockedAmount.add(amount).mul(10).div(totalBalance())< 8)` (line 115).

```
108     /*
109      * @nonce calls by HegicCallOptions to lock the funds
110      * @param amount Amount of funds that should be locked in an option
111      */
112     function lock(uint id, uint256 amount) external override onlyOwner payable {
113         require(id == lockedLiquidity.length, "Wrong id");
114         require(
115             lockedAmount.add(amount).mul(10).div(totalBalance()) < 8,
116             "Pool Error: Amount is too large."
117         );
118 
119         lockedLiquidity.push(LockedLiquidity(amount, msg.value, true));
120         lockedPremium = lockedPremium.add(msg.value);
121         lockedAmount = lockedAmount.add(amount);
122     }
```

Listing 3.10: HegicETHPool.sol

The use of division in line 115 may inevitably introduce a (small) precision loss. To remedy that, a better approach is to change the requirement into the following one: `require(lockedAmount.add(amount).mul(10))< totalBalance().mul(8))`. By doing so, we can ensure there is no precision loss in this particular case.

Recommendation Revise the threshold enforcement of locked amount in the pool to avoid any unnecessary precision loss.

```
    /*
     * @nonce calls by HegicCallOptions to lock the funds
     * @param amount Amount of funds that should be locked in an option
     */
    function lock(uint id, uint256 amount) external override onlyOwner payable {
        require(id == lockedLiquidity.length, "Wrong id");
        require(
            lockedAmount.add(amount).mul(10) <= totalBalance().mul(8),
            "Pool Error: Amount is too large."
        );

        lockedLiquidity.push(LockedLiquidity(amount, msg.value, true));
        lockedPremium = lockedPremium.add(msg.value);
        lockedAmount = lockedAmount.add(amount);
    }
```

Listing 3.11: HegicETHPool.sol

Status This issue has been fixed in the commit: 83499168bbbf622cae53527e49576e340d06be8c.

3.5 Improved Precision With Ceiling Division

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: HegicETHPool/HegicERCPool
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [3]

Description

SafeMath is a Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the default division behavior, i.e., the floor division.

Conceptually, the floor division is a normal division operation except it returns the largest possible integer that is either less than or equal to the normal division result. In SafeMath, floor(x) or simply div takes as input an integer number x and gives as output the greatest integer less than or equal to x, denoted floor(x) $= \lfloor x \rfloor$. Its counterpart is the ceiling division that maps x to the least integer greater than or equal to x, denoted as ceil(x) $= \lceil x \rceil$. In essence, the ceiling division is rounding up the result of the division, instead of rounding down in the floor division.

As examined in Section 3.2, a Hegic pool has defined two main APIs for its management: `provide()` and `withdraw()`. The `provide()` routine is used to add funds into the pool while the `withdraw()` routine is used to withdraw funds from the pool.

During the analysis of `withdraw()`, we notice the burn amount calculation results in (small) precision loss. For elaboration, we show the related code snippet below.

```
83     /*
84      * @nonce Provider burns writeETH and receives ETH from the pool
85      * @param amount Amount of ETH to receive
86      * @return burn Amount of tokens to be burnt
87      */
88     function withdraw(uint256 amount, uint256 maxBurn) external returns (uint256 burn) {
89         require(
90             lastProvideTimestamp[msg.sender].add(lockupPeriod) <= block.timestamp,
91             "Pool: Withdrawal is locked up"
92         );
93         require(
94             amount <= availableBalance(),
95             "Pool Error: Not enough funds on the pool contract. Please lower the amount.
                   "
96         );
97         burn = amount.mul(totalSupply()).div(totalBalance());
98 
99         require(burn <= maxBurn, "Pool: Burn limit is too small");
100        require(burn <= balanceOf(msg.sender), "Pool: Amount is too large");
101        require(burn > 0, "Pool: Amount is too small");
102 
103        _burn(msg.sender, burn);
104        emit Withdraw(msg.sender, amount, burn);
105        msg.sender.transfer(amount);
106    }
```

Listing 3.12: HegicETHPool.sol

Specifically, the burn amount is calculated as `burn = amount.mul(totalSupply()).div(totalBalance())` (line 97). Apparently, it is a standard `floor()` operation that rounds down the calculation result. Note that in a pool scenario where a liquidity provider wants to withdraw previous deposits, if there is a rounding issue, it is always preferable to calculate the trading amount in a way towards the liquidity pool. Therefore, depending on specific cases, the calculation may often needs to replace the normal `floor` division with `ceiling` division. In other words, the burn amount calculation is better revised as `burn = amount.mul(totalSupply()).sub(1).div(totalBalance()).add(1)`, a `ceiling` division.

Recommendation Revise the logic accordingly to round-up the burn amount calculation. Note both pool contracts (`HegicETHPool` and `HegicERCPool`) share the same issue.

```
    /*
     * @nonce Provider burns writeETH and receives ETH from the pool
     * @param amount Amount of ETH to receive
     * @return burn Amount of tokens to be burnt
```

```
    */
    function withdraw(uint256 amount, uint256 maxBurn) external returns (uint256 burn) {
        require(
            lastProvideTimestamp[msg.sender].add(lockupPeriod) <= block.timestamp,
            "Pool: Withdrawal is locked up"
        );
        require(
            amount <= availableBalance(),
            "Pool Error: Not enough funds on the pool contract. Please lower the amount."
        );
        burn = amount.mul(totalSupply()).sub(1).div(totalBalance()).add(1);

        require(burn <= maxBurn, "Pool: Burn limit is too small");
        require(burn <= balanceOf(msg.sender), "Pool: Amount is too large");
        require(burn > 0, "Pool: Amount is too small");

        _burn(msg.sender, burn);
        emit Withdraw(msg.sender, amount, burn);
        msg.sender.transfer(amount);
    }
```

Listing 3.13: HegicETHPool.sol

Status This issue has been fixed in the commit: 1289891a39e06a865ec7d932c006e466afbed006.

3.6 Less Friction For Option Creation

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `HegicETHOptions`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [2]

Description

Hegic has a number of components that not only depend on each other, but also interact with external DeFi protocols. Because of that, it is often necessary to introduce as little friction as possible to avoid sudden disruption of an ongoing transaction. Note that the disruption can be caused by imposed requirements on the related execution paths. Certainly, essential requirements need to be satisfied while others need to gauge specific application situations or logics to avoid unnecessary or sudden `revert`.

In the following, we show a specific case in the `HegicETHOptions` contract. The specific function is `create()` that is used to create a new option. Note this routine performs a number of sanity checks, including the option periods as well as related fee requirements. The particular statement of `require(msg.value == total, "Wrong value")` (line 117) asks for the exact `ETH` amount being transferred in. This requirement will unnecessarily revert the ongoing transaction even if the routine receives enough payment.

```
98      function create(
99          uint256 period,
100         uint256 amount,
101         uint256 strike,
102         OptionType optionType
103     )
104         external
105         payable
106         returns (uint256 optionID)
107     {
108         (uint256 total, uint256 settlementFee, uint256 strikeFee, ) = fees(
109             period,
110             amount,
111             strike,
112             optionType
113         );
114         require(period >= 1 days, "Period is too short");
115         require(period <= 4 weeks, "Period is too long");
116         require(amount > strikeFee, "Price difference is too large");
117         require(msg.value == total, "Wrong value");
118
119         uint256 strikeAmount = amount.sub(strikeFee);
120         optionID = options.length;
121         Option memory option = Option(
122             State.Active,
123             msg.sender,
124             strike,
125             amount,
126             strikeAmount.mul(optionCollateralizationRatio).div(100).add(strikeFee),
127             total.sub(settlementFee),
128             block.timestamp + period,
129             optionType
130         );
131
132         options.push(option);
133         settlementFeeRecipient.sendProfit {value: settlementFee}();
134         pool.lock {value: option.premium} (optionID, option.lockedAmount);
135         emit Create(optionID, msg.sender, settlementFee, total);
136     }
```

Listing 3.14: HegicETHOptions.sol

A more graceful approach is to allow for a larger payment, but only take the needed amount and then return extra back to the sender. By doing so, we avoid introducing unnecessary frictions.

Recommendation Introduce as little friction as possible by revising the `create()` routine accordingly.

```
    function create(
        uint256 period,
        uint256 amount,
        uint256 strike,
        OptionType optionType
    )
        external
        payable
        returns (uint256 optionID)
    {
        (uint256 total, uint256 settlementFee, uint256 strikeFee, ) = fees(
            period,
            amount,
            strike,
            optionType
        );
        require(period >= 1 days, "Period is too short");
        require(period <= 4 weeks, "Period is too long");
        require(amount > strikeFee, "Price difference is too large");
        require(msg.value >= total, "Wrong value");
        if (msg.value > total)
            msg.sender.transfer(msg.value - total);

        uint256 strikeAmount = amount.sub(strikeFee);
        optionID = options.length;
        Option memory option = Option(
            State.Active,
            msg.sender,
            strike,
            amount,
            strikeAmount.mul(optionCollateralizationRatio).div(100).add(strikeFee),
            total.sub(settlementFee),
            block.timestamp + period,
            optionType
        );

        options.push(option);
        settlementFeeRecipient.sendProfit {value: settlementFee}();
        pool.lock {value: option.premium} (optionID, option.lockedAmount);
        emit Create(optionID, msg.sender, settlementFee, total);
    }
```

Listing 3.15: HegicETHOptions.sol

Status This issue has been fixed in the commit: 83499168bbbf622cae53527e49576e340d06be8c. While reviewing this particular commit, we notice the related overpayment is returned twice and this has been accordingly fixed in this commit: 1f344462d1f3a501ec20fbcecc7ae697bc43c2a0.

3.7 Wrong Reward Rate in HegicWBTCRewards

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact:Medium
- Target: HegicWBTCRewards
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [3]

Description

The Hegic protocol has defined two reward contracts, i.e., HegicETHRewards and HegicWBTCRewards. Both inherit from the base contract of HegicETHRewards which has the following constructor and variables.

```
abstract
contract HegicRewards is Ownable {
    using SafeMath for uint;
    using SafeERC20 for IERC20;

    IHegicOptions public immutable hegicOptions;
    IERC20 public immutable hegic;
    mapping(uint => bool) public rewardedOptions;
    mapping(uint => uint) public dailyReward;
    uint internal constant MAX_DAILY_REWARD = 165_000e18;
    uint internal constant REWARD_RATE_ACCURACY = 1e8;
    uint internal immutable MAX_REWARDS_RATE;
    uint internal immutable MIN_REWARDS_RATE;
    uint public rewardsRate;

    constructor(
        IHegicOptions _hegicOptions,
        IERC20 _hegic,
        uint maxRewardsRate,
        uint minRewardsRate
    ) public {
        hegicOptions = _hegicOptions;
        hegic = _hegic;
        MAX_REWARDS_RATE = maxRewardsRate;
        MIN_REWARDS_RATE = minRewardsRate;
        rewardsRate = maxRewardsRate;
    }
```

Listing 3.16: The HegicRewards Contract

We notice the REWARD_RATE_ACCURACY = 1e8, which implies that reward rate has the decimal of 8. However, when the HegicWBTCRewards contract instantiates HegicRewards, we observe the following instantiation.

```
contract HegicWBTCRewards is HegicRewards {
```

```
    constructor(
        IHegicOptions _hegicOptions,
        IERC20 _hegic
    ) public HegicRewards(
        _hegicOptions,
        _hegic,
        1_000_000e18,
        10e18
    ) {}
}
```

Listing 3.17: The HegicWBTCRewards Contract

In particular, `maxRewardsRate` and `minRewardsRate` are initialized as `1_000_000e18` and `10e18`, respectively. Apparently, they are assuming the decimal of 18, not 8. The decimal mismatch could result in immediate depletion of available rewards for distribution.

Recommendation Correct the above-mentioned decimal mismatch in `HegicWBTCRewards`. Note that `HegicETHRewards` is not affected.

```
contract HegicWBTCRewards is HegicRewards {
    constructor(
        IHegicOptions _hegicOptions,
        IERC20 _hegic
    ) public HegicRewards(
        _hegicOptions,
        _hegic,
        1_000_000e8,
        10e8
    ) {}
}
```

Listing 3.18: The HegicWBTCRewards Contract

Status This issue has been fixed in the commit: 1f344462d1f3a501ec20fbcecc7ae697bc43c2a0.

3.8 Suggested Reservation of The First enum Element

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `IHegicOptions`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [2]

Description

The Solidity language supports that `enum` type that allows to create a user-defined type. They are explicitly convertible to and from all integer types, but implicit conversion is not allowed. The explicit conversions check the value ranges at runtime and a failure causes an exception. The `enum` type needs at least one member and all members are represented by subsequent unsigned integer values starting from 0, in the order they are defined.

If we examine the option state and type defined in `IHegicOptions`, there are three option states, i.e., `Active`, `Exercised`, and `Expired`. and two option types, i.e., `Put` and `Call`. Note that the first `enum` member equals the number 0, which coincidentally is the same as the default or uninitialized integer. Because of that, we strongly suggest to have the first `enum` member as a placeholder member to avoid any unnecessary misinterpretation.

```
interface IHegicOptions {
    event Create(
        uint256 indexed id,
        address indexed account,
        uint256 settlementFee,
        uint256 totalFee
    );

    event Exercise(uint256 indexed id, uint256 profit);
    event Expire(uint256 indexed id, uint256 premium);
    enum State {Active, Exercised, Expired}
    enum OptionType {Put, Call}

    ...

}
```

Listing 3.19: IHegicOptions.sol

Using the `enum State` as an example, if we consider the public `unlock()` function, we may provide an arbitrary large `optionID`. This corresponding option does not exist. However, it can successfully pass the sanity checks performed in lines 221-222. The fact that the specific check `option.state == State.Active` is met should be alarming for an non-exist option! Fortunately, the compiler will generate an implicit bound-check for an array so that the option index stays within the array range `[0, options.length-1]`. Nevertheless, the misinterpretation of the first `enum` member as 0 needs to be avoided as much as possible.

```
215     /**
216      * @notice Unlock funds locked in the expired options
217      * @param optionID ID of the option
218      */
219     function unlock(uint256 optionID) public {
220         Option storage option = options[optionID];
221         require(option.expiration < block.timestamp, "Option has not expired yet");
222         require(option.state == State.Active, "Option is not active");
223         option.state = State.Expired;
224         pool.unlock(optionID);
225         emit Expire(optionID, option.premium);
```

```
    }
```

Listing 3.20: HegicETHOptions.sol

Recommendation Revised the defined `enum` members as the following:

```
interface IHegicOptions {
    event Create(
        uint256 indexed id,
        address indexed account,
        uint256 settlementFee,
        uint256 totalFee
    );

    event Exercise(uint256 indexed id, uint256 profit);
    event Expire(uint256 indexed id, uint256 premium);
    enum State {Inactive, Active, Exercised, Expired}
    enum OptionType {Invalid, Put, Call}

    ...

}
```

Listing 3.21: IHegicOptions.sol

Status This issue has been fixed in the commit: 83499168bbbf622cae53527e49576e340d06be8c.

3.9 Enhanced Business Logic of lock() in HegicETHPool

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HegicETHPool`
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

As discussed in Section 3.4, the Hegic protocol takes a rather prudent approach in maintaining a threshold of 80% of locked funds above which no new option will be created. This restriction is enforced when a new option always needs to lock certain amount of funds in the pool (in the `lock()` routine as shown below), i.e., `require(lockedAmount.add(amount).mul(10).div(totalBalance())< 8)` (line 115).

```
108     /*
109      * @nonce calls by HegicCallOptions to lock the funds
```

```
     * @param amount Amount of funds that should be locked in an option
     */
    function lock(uint id, uint256 amount) external override onlyOwner payable {
        require(id == lockedLiquidity.length, "Wrong id");
        require(
            lockedAmount.add(amount).mul(10).div(totalBalance()) < 8,
            "Pool Error: Amount is too large."
        );

        lockedLiquidity.push(LockedLiquidity(amount, msg.value, true));
        lockedPremium = lockedPremium.add(msg.value);
        lockedAmount = lockedAmount.add(amount);
    }
```

Listing 3.22: HegicETHPool.sol

We have taken a further analysis on the threshold calculation and our result shows that the calculation of `HegicETHPool` can be more precise. Specifically, the current denominator is `totalBalance()`, which is essentially calculated as `address(this).balance.sub(lockedPremium)` (line 191).

In `HegicETHPool`, the `lock()` routine is defined as `payable`, which means the `totalBalance()` in the equation has already taken into account the accompanying `msg.value` that is just transferred in. With the further consideration of PVE-004, we need to revise the threshold as follows: `require(lockedAmount.add(amount).mul(10)< totalBalance().sub(msg.value).mul(8))`.

Recommendation Revise the threshold enforcement of locked amount in `HegicETHPool` for improved accuracy.

```
    /*
     * @nonce calls by HegicCallOptions to lock the funds
     * @param amount Amount of funds that should be locked in an option
     */
    function lock(uint id, uint256 amount) external override onlyOwner payable {
        require(id == lockedLiquidity.length, "Wrong id");
        require(
            lockedAmount.add(amount).mul(10) < totalBalance().sub(msg.value).mul(8),
            "Pool Error: Amount is too large."
        );

        lockedLiquidity.push(LockedLiquidity(amount, msg.value, true));
        lockedPremium = lockedPremium.add(msg.value);
        lockedAmount = lockedAmount.add(amount);
    }
```

Listing 3.23: HegicETHPool.sol

Status This issue has been fixed in the commit: 83499168bbbf622cae53527e49576e340d06be8c.

3.10 Redundant lockupFree Verification

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `HegicStaking`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [4]

Description

As mentioned in Section 3.1, the Hegic protocol will generate and collect settlement fees (in `ETH` and `WBTC`) paid every time when a hegic option contract is purchased. The `HEGIC` token holders can stake their tokens to receive pro-rata staking rewards. In order to prevent possible flashloan-assisted front-running attacks that may claim the majority of rewards, the staking logic is designed to have a lockup period for staked assets. For each account, the associated lockup period is recorded as `[lastBoughtTimestamp[account], lastBoughtTimestamp[account].add(lockupPeriod)]`.

When analyzing the unlocking logic of staked assets, we notice there is a redundant validity check on the lockup period. Specifically, we show below the `sell()` logic behind the unlocking logic. The modifier `lockupFree` essentially enforces the same requirement as specified at line 76. With that, we can safely remove one without weakening the needed enforcement.

```
74    function sell(uint amount) external override lockupFree {
75        require(
76            lastBoughtTimestamp[msg.sender].add(lockupPeriod) <= block.timestamp
77        );
78        _burn(msg.sender, amount);
79        HEGIC.safeTransfer(msg.sender, amount.mul(LOT_PRICE));
80    }
```

Listing 3.24: HegicStaking.sol

Recommendation Consider the removal of the redundant verification as follows:

```
    function sell(uint amount) external override lockupFree {
        _burn(msg.sender, amount);
        HEGIC.safeTransfer(msg.sender, amount.mul(LOT_PRICE));
    }
```

Listing 3.25: HegicStaking.sol

Status This issue has been fixed in the commit: a11349afc3585377dd02910f0a2ff8d34b926385.

3.11 Denial-of-Service in getReward()

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `HegicRewards`
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

Following the discussions of Section 3.3, we further examine the incentivization mechanism to encourage early adoption. Specifically, each option holder is eligible to claim the rewards via the `getReward()` routine. The logic is rather straightforward in firstly determining the reward amount, next marking the option's reward-claiming status, then ensuring the rewarded amount still falls within the daily rewarding limit, and finally transferring the reward.

Our analysis shows that the above logic does not validate the given `optionID`. Because of that, a malicious actor may submit a `getReward()` request with an `optionID` that has not been created yet (but is expected to be created soon). Considering the current algorithm for `optionID` assignment, any one can reliably guess the next `optionID` to be created. Consequently, the owner of new `optionID` will be unable to receive the reward.

```
    function getReward(uint optionId) external {
        uint amount = rewardAmount(optionId);
        uint today = block.timestamp / 1 days;
        (, address holder, , , , , ) = hegicOptions.options(optionId);
        require(!rewardedOptions[optionId], "The option was rewarded");
        require(
            amount.add(dailyReward[today]) < MAX_DAILY_REWARD,
            "Exceeds daily limits"
        );
        rewardedOptions[optionId] = true;
        hegic.safeTransfer(holder, amount);
    }
```

Listing 3.26: HegicRewards.sol

Recommendation Apply necessary sanity checks in `getReward()` to prevent invalid options from claiming rewards. An example revision is shown in the following:

```
    function getReward(uint optionId) external {
        uint amount = rewardAmount(optionId);
        uint today = block.timestamp / 1 days;
        (IHegicOptions.State state, address holder, , , , , ) = hegicOptions.options(
            optionId);
        require(state != IHegicOptions.State.Inactive, "The option is inactive");
        require(!rewardedOptions[optionId], "The option was rewarded");
```

```
        require(
            amount.add(dailyReward[today]) < MAX_DAILY_REWARD,
            "Exceeds daily limits"
        );
        rewardedOptions[optionId] = true;
        dailyReward[today] = dailyReward[today].add(amount);
        hegic.safeTransfer(holder, amount);
    }
```

Listing 3.27: HegicRewards.sol

Status This issue has been fixed in the commit: 1f344462d1f3a501ec20fbcecc7ae697bc43c2a0.

3.12 Option Pool Draining With Invalid optionType

- ID: PVE-012
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: HegicETHOptions/HegicWBTCOptions
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

Hegic options are created with four required elements, i.e., `period`, `amount`, `strike`, and `optionType`. These four elements are essential in calculating respective fees (e.g., premium, locked assets, and settlement fee) and then introducing a new option into the protocol.

To elaborate, we show below the `create()` routine of the `HegicETHOptions` contract. Note this routine performs a number of sanity checks, including the option periods as well as related fee requirements. However, it does not validate the last parameter `optionType`. With that, a malicious actor could potentially craft a new option with an invalid `optionType` to drain all available funds in the pool.

```
    function create(
        uint256 period,
        uint256 amount,
        uint256 strike,
        OptionType optionType
    )
        external
        payable
        returns (uint256 optionID)
    {
        (uint256 total, uint256 settlementFee, uint256 strikeFee, ) = fees(
            period,
```

```
            amount,
            strike,
            optionType
        );
        require(period >= 1 days, "Period is too short");
        require(period <= 4 weeks, "Period is too long");
        require(amount > strikeFee, "Price difference is too large");
        require(msg.value == total, "Wrong value");

        uint256 strikeAmount = amount.sub(strikeFee);
        optionID = options.length;
        Option memory option = Option(
            State.Active,
            msg.sender,
            strike,
            amount,
            strikeAmount.mul(optionCollateralizationRatio).div(100).add(strikeFee),
            total.sub(settlementFee),
            block.timestamp + period,
            optionType
        );

        options.push(option);
        settlementFeeRecipient.sendProfit {value: settlementFee}();
        pool.lock {value: option.premium} (optionID, option.lockedAmount);
        emit Create(optionID, msg.sender, settlementFee, total);
    }
```

Listing 3.28: HegicETHOptions.sol

Specifically, a malicious actor requests a new option with the four required elements: `period = 1 days`, `amount = 1 eth`, `strike = latestPrice*10**18`, and `optionType = 100`. These elements can successfully pass current sanity checks and result in a new option creation with the following respective fees: `settlementFee = 0.1 eth`, `strikeFee = 0 eth` and `protocolFee = amount*sqrt(period)*impliedVolRate/10**26 ~= 0`.

After the creation, the crafted option can be immediately exercised and the main logic is implemented in the `payProfit()` routine (shown below). As the option's `optionType` is crafted, which is not `OptionType.Call`, the routine takes the `else` branch in lines 314 – 317. Also, since the `strike` price is significantly larger than the `latestPrice`, the resulting `profit` (line 316) becomes significantly larger than the option's locked amount. As a result, the malicious actor can immediately exercise the crafted option to get back the the option's locked amount (line 320).

```
300    /**
301     * @notice Sends profits in ETH from the ETH pool to an option holder's address
302     * @param optionID A specific option contract id
303     */
304    function payProfit(uint optionID)
305        internal
306        returns (uint profit)
```

```
    {
        Option memory option = options[optionID];
        (, int latestPrice, , , ) = priceProvider.latestRoundData();
        uint256 currentPrice = uint256(latestPrice);
        if (option.optionType == OptionType.Call) {
            require(option.strike <= currentPrice, "Current price is too low");
            profit = currentPrice.sub(option.strike).mul(option.amount).div(currentPrice
                );
        } else {
            require(option.strike >= currentPrice, "Current price is too high");
            profit = option.strike.sub(currentPrice).mul(option.amount).div(currentPrice
                );
        }
        if (profit > option.lockedAmount)
            profit = option.lockedAmount;
        pool.send(optionID, option.holder, profit);
    }
```

Listing 3.29: HegicETHOptions.sol

To summarize, the actor essentially invests `1% * amount` into the option creation, but immediately gets back the corresponding locked amount, i.e., `amount.mul(optionCollateralizationRatio).div(100) = 50% * amount`. By continuing the above process, the actor can drain all funds available in the current pool. Note both `HegicETHOptions` and `HegicWBTCOptions` are affected.

Recommendation Validate the given `optionType` in both pools and prevent invalid ones from entering the option creation.

Status This issue has been fixed in the commit: a11349afc3585377dd02910f0a2ff8d34b926385.

4 | Conclusion

In this audit, we thoroughly analyzed the Hegic design and implementation. The system presents a unique offering in current DeFi ecosystem in allowing for trustless, non-custodial creation, maintenance, and settlement of hedge contracts. 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [11, 12, 13, 14, 16].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [17] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

[11] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[12] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[13] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[14] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[16] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[17] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.