SMART CONTRACT AUDIT REPORT

for

CVI

Prepared By: Shuxiao Wang

PeckShield

May 30, 2021

Document Properties

Client	Coti
Title	Smart Contract Audit Report
Target	CVI
Version	1.0
Author	Xuxian Jiang
Auditors	Jian Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 30, 2021	Xuxian Jiang	Final Release
1.0-rc	May 20, 2021	Xuxian Jiang	Release Candidate
0.2	May 11, 2021	Xuxian Jiang	Add More Findings #1
0.1	May 4, 2021	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 **CVI Protocol** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is well-designed. This document outlines our audit results.

1.1 About CVI Protocol

The `Cryptocurrency Volatility Index (CVI)` is a full-scale decentralized ecosystem that brings the sophisticated and very popular "market fear index" to the crypto market and is created by computing a decentralized volatility index from cryptocurrency option prices, together with analyzing the market's expectation of future volatility. It allows for analyzing the market's expectation of future volatility and enabling users to open positions based on these predictions that can be subsequently liquidated or redeemed depending on whether the value remains above the liquidation threshold.

The basic information of the CVI protocol is as follows:

Table 1.1: Basic Information of CVI

Item	Description
Name	Coti
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 30, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited repository contains a number of sub-directories (e.g., `v1`, `v2`, and `v3`) and this audit covers only the `v4` sub-directory and associated dependencies.

- https://github.com/cotitech-io/cvi-contracts.git (ecb86db)

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

- https://github.com/cotitech-io/cvi-contracts.git (d84a199)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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			
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low
	High	*Medium*	*Low*
		Likelihood	

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [11]:

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

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

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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the CVI 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	0	
High	1	🟦
Medium	3	🟦🟦🟦
Low	3	🟦🟦🟦
Informational	0	🟦
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerability, and 3 low-severity vulnerabilities.

Table 2.1: Key CVI Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Incorrect delta Calculation in rebaseCVI()	Business Logic	Resolved
PVE-002	Low	Incorrect Transfer Event For ElasticToken::_burn()	Code Practices	Resolved
PVE-003	Low	Improved Sanity Checks Of System/Function Parameters	Code Practices	Resolved
PVE-004	Medium	Possible Front-Running/MEV For Reduced Conversion	Time And State	Resolved
PVE-005	Low	Improved Support in ETHVolatility	Business Logic	Resolved
PVE-006	Medium	Premium Fee Avoidance in ETHPlatformV2	Security Features	Resolved
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Incorrect delta Calculation in rebaseCVI()

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

Description

The `CVI` token is a rebasing one that will be periodically adjusted. The core logic is mainly implemented in the `VolatilityToken` contract. While examining the rebasing logic, we notice the calculation of index delta for adjustment is flawed.

To elaborate, we show below the `rebaseCVI()` routine. This routine can be invoked after the associated `uniswapOracle` is updated. The main logic is to compute the price deviation (line 73) between `uniswapOracle` and `cviOracle`. The rebasing operation will not proceed until the computed devision is larger than the specified `minDeviation` (line 77). However, the `delta` calcuation is incorrect as the current formula of `delta = DELTA_PRECISION_DECIMALS.mul(cviValue).div(deviation).div(rebaseLag)` should be replace with the following one: `DELTA_PRECISION_DECIMALS.mul(deviation).div(cviValue).div(rebaseLag)`.

```
66      // If not rebaser, the rebase underlying method will revert
67      function rebaseCVI() external override {
68          require(uniswapOracle.blockTimestampLast() + uniswapOracle.PERIOD() >= block.
                timestamp, "Price not updated");
69
70          (uint256 cviValue,,) = cviOracle.getCVILatestRoundData();
71          uint256 uniswapValue = uniswapOracle.consult(address(this), 1);
72
73          uint256 deviation = uniswapValue > cviValue ? uniswapValue - cviValue : cviValue
              - uniswapValue;
74          bool positive = uniswapValue > cviValue;
75
76        uint256 minDeviation = cviValue.mul(minDeviationPercentage).div(MAX_PERCENTAGE);
77        require(deviation >= minDeviation, "Not enough deviation");
78
79        uint256 delta = DELTA_PRECISION_DECIMALS.mul(cviValue).div(deviation).div(
              rebaseLag);
80        rebase(delta, positive);
81    }
```

Listing 3.1: VolatilityToken ::rebaseCVI()

Recommendation Revise the above `rebaseCVI()` logic to properly compute the index delta for rebasing.

Status This issue has been fixed in this commit: `a146a46`.

3.2 Incorrect Transfer Event For ElasticToken::_burn()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ElasticToken`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `ElasticToken` contract as an example. This contract is designed to support the rebasing operation. While examining the events that reflect the token dynamics, we notice the `Transfer` event is not properly emitted when the token is being burned.

To elaborate, we show below its `_burn()` routine, which is designed to burn given amount of tokens. Note that it indeed emits the ERC20-compliant `Transfer` event for the burn operation. However, it does not properly encode the emitted content. Specifically, it is currently emitted as `Transfer(address(0), to, amount)` (line 77), where the source address should be `to`, while the destination should be `address(0)`.

```
66    function _burn(address to, uint256 amount) internal validRecipient(to) {
67        _beforeTokenTransfer(to, address(0), amount);
68
69        totalSupply = totalSupply.sub(amount);
```

```
        uint256 underlyingValue = valueToUnderlying(amount);

        // Note: as initSupply decreases, max sacling factor increases, so no need to
            test scaling factor against it
        initSupply = initSupply.sub(underlyingValue);

        _underlyingBalances[to] = _underlyingBalances[to].sub(underlyingValue, "Burn
            amount exceeds balance");

        emit Transfer(address(0), to, amount);
    }
```

Listing 3.2: ElasticToken :: _burn()

Recommendation Properly emit the `Transfer` event in all cases. This is very helpful for external analytics and reporting tools.

Status This issue has been fixed in this commit: `a146a46`.

3.3 Improved Sanity Checks For System/Function Parameters

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

Description

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

```
    function setTimePenaltyFeeParameters(uint16 _minTimePenaltyFeePercent, uint32
        _midTime, uint16 _midTimePenaltyFeePercent, uint32 _maxTime, uint16
        _maxTimePenaltyFeePercent) external override onlyOwner {
        require(_minTimePenaltyFeePercent <= MAX_FEE_PERCENTAGE, "Min fee larger than
            max fee");
        require(_midTimePenaltyFeePercent <= MAX_FEE_PERCENTAGE, "Mid fee larger than
            max fee");
        require(_maxTimePenaltyFeePercent <= MAX_FEE_PERCENTAGE, "Max fee larger than
            max fee");
        require(_midTime <= _maxTime, "Max time before mid time");
        require(_midTimePenaltyFeePercent <= _maxTimePenaltyFeePercent, "Max fee less
            than mid fee");

        minTimePenaltyFeePercent = _minTimePenaltyFeePercent;
        midTime = _midTime;
        midTimePenaltyFeePercent = _midTimePenaltyFeePercent;
        maxTime = _maxTime;
        maxTimePenaltyFeePercent = _maxTimePenaltyFeePercent;
    }
```

Listing 3.3: RequestFeesCalculator :: setTimePenaltyFeeParameters()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely misconfiguration of `minTimePenaltyFeePercent` `midTimePenaltyFeePercent`, and `maxTimePenaltyFeePercent` may introduce an unreasonably fee order. Specifically, there is a need to enforce `_minTimePenaltyFeePercent` is not larger than `_midTimePenaltyFeePercent`, i.e., `require(_minTimePenaltyFeePercent <= _midTimePenalty FeePercent, "Mid fee less than min fee");`

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

Status This issue has been fixed in this commit: `a146a46`.

3.4 Potential Front-Running/MEV With Reduced Conversion

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StakingV2`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

The `CVI` protocol has a built-in staking contract `StakingV2` to engage user participation and community. The contract supports two types of reward tokens: `claimableTokens` and `otherTokens`. The first type allows users to directly claim by themselves, while the second type will be converted into `WETH` as profit and then disseminated to staking users. To elaborate, we show below the `convertFunds()` routine that is designed to convert credited `otherTokens` into `WETH`.

```
    function convertFunds() external override {
        bool didConvert = false;
        for (uint256 tokenIndex = 0; tokenIndex < otherTokens.length; tokenIndex++) {
            IERC20 token = otherTokens[tokenIndex];
            uint256 balance = token.balanceOf(address(this));

            if (balance > 0) {
                didConvert = true;

                uint256 amountSwapped = swapper.swapToWETH(token, token.balanceOf(
                    address(this)));
                addProfit(amountSwapped, IERC20(address(wethToken)));
            }
        }

        require(didConvert, "No funds to convert");
    }
```

Listing 3.4: StakingV2::convertFunds()

We notice the collected yields are routed to `swapper`, i.e., `UniswapV2`, in order to swap them to `WETH` as profit. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user or the `staking` contract in our case because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status This issue has been fixed in this commit: `a146a46`.

3.5 Improved Support in ETHVolatility

- ID: PVE-005
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `ETHVolatility`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

To provide native ETH support, the CVI protocol provides convenient contracts ETHVolatilityToken and ETHPlatformV2. The first one allows for direct ETH deposits as liquidity providers or position buyers. However, our analysis shows the ETH support in ETHVolatilityToken can be further improved.

To elaborate, we show below the constructor routine of ETHVolatilityToken. The constructor routine relies on the VolatilityToken's constructor routine, which is also shown below.

```
contract ETHVolatilityToken is VolatilityToken, IETHVolatilityToken {

    IETHPlatformV2 public ethPlatform;
    address payable public feesCollectorAddress;

    constructor(string memory _lpTokenName, string memory _lpTokenSymbolName, uint8
        _leverage, uint256 _initialTokenToLPTokenRate,
        IETHPlatformV2 _ethPlatform, address payable _feesCollectorAddress,
            IFeesCalculatorV4 _feesCalculator, IRequestFeesCalculator
            _requestFeesCalculator, ICVIOracleV3 _cviOracle, IUniswapOracle
            _uniswapOracle)
            VolatilityToken (IERC20(address(0)), _lpTokenName, _lpTokenSymbolName,
                _leverage, _initialTokenToLPTokenRate, IPlatformV3(address(0)),
                IFeesCollector(address(0)), _feesCalculator, _requestFeesCalculator,
                _cviOracle, _uniswapOracle) {
        ethPlatform = _ethPlatform;
        feesCollectorAddress = _feesCollectorAddress;
    }
    ...
}
```

Listing 3.5: ETHVolatilityToken::constructor()

```
    constructor(IERC20 _token, string memory _lpTokenName, string memory
        _lpTokenSymbolName, uint8 _leverage, uint256 _initialTokenToLPTokenRate,
            IPlatformV3 _platform, IFeesCollector _feesCollector, IFeesCalculatorV4
                _feesCalculator, IRequestFeesCalculator _requestFeesCalculator,
                ICVIOracleV3 _cviOracle, IUniswapOracle _uniswapOracle) ElasticToken (
                _lpTokenName, _lpTokenSymbolName, 18) {
        token = _token;
        platform = _platform;
        feesCollector = _feesCollector;
        feesCalculator = _feesCalculator;
        requestFeesCalculator = _requestFeesCalculator;
        cviOracle = _cviOracle;
        uniswapOracle = _uniswapOracle;
        initialTokenToLPTokenRate = _initialTokenToLPTokenRate;
        leverage = _leverage;

        token.approve(address(_platform), uint256(-1));
        token.approve(address(_feesCollector), uint256(-1));
    }
```

Listing 3.6: VolatilityToken ::constructor()

We notice that within the `VolatilityToken`'s constructor, there are two `approve()` statements (lines $62-63$). While it is reasonable for regular ERC20 tokens, it is not applicable for `ETHVolatilityToken`, which has `IERC20(address(0)` as the `token`.

Recommendation Revise the constructor logic of `VolatilityToken` to avoid `approve()` calls for the `ETH` support.

Status This issue has been fixed in this commit: `a146a46`.

3.6 Premium Fee Avoidance in ETHPlatformV2

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ETHPlatformV2`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

As mentioned in Section 3.5, to provide native `ETH` support, the `CVI` protocol provides convenient contracts `ETHVolatilityToken` and `ETHPlatformV2`. Also, we notice the `PlatformV3` contract allows certain users from being charged for the premium fee. However, the convenience contract `ETHVolatilityToken` may be leveraged to completely bypass the premium fee.

In the following, we show the related two routines: `openPositionETH()` and `openPositionWithoutPremiumFeeETH()`. Both routines invoke the `_openPosition()` helper to open a position with the premium fee turned off and on respectively. However if we examine the `openPositionWithoutPremiumFee()` implementation from `PlatformV3`, it has the proper validation on whether the user has been exempted from the premium fee.

```
    function openPositionETH(uint16 _maxCVI, uint168 _maxBuyingPremiumFeePercentage,
        uint8 _leverage) external override payable nonReentrant returns (uint168
        positionUnitsAmount, uint168 positionedETHAmount) {
        require(uint168(msg.value) == msg.value, "Too much ETH");
        return _openPosition(uint168(msg.value), _maxCVI, _maxBuyingPremiumFeePercentage
            , _leverage, true);
    }

    function openPositionWithoutPremiumFeeETH(uint16 _maxCVI, uint168
        _maxBuyingPremiumFeePercentage, uint8 _leverage) external override payable
        returns (uint168 positionUnitsAmount, uint168 positionedTokenAmount) {
        require(uint168(msg.value) == msg.value, "Too much ETH");
        return _openPosition(uint168(msg.value), _maxCVI, _maxBuyingPremiumFeePercentage
            , _leverage, false);
    }
```

Listing 3.7: ETHPlatformV2::openPositionETH()/openPositionWithoutPremiumFeeETH()

Recommendation Revised the `openPositionETH()` logic for proper user validation. An example revision is shown below.

```
    function openPositionETH(uint16 _maxCVI, uint168 _maxBuyingPremiumFeePercentage,
        uint8 _leverage) external override payable nonReentrant returns (uint168
        positionUnitsAmount, uint168 positionedETHAmount) {
        require(noPremiumFeeAllowedAddresses[msg.sender], "Not allowed");
        require(uint168(msg.value) == msg.value, "Too much ETH");
        return _openPosition(uint168(msg.value), _maxCVI, _maxBuyingPremiumFeePercentage
            , _leverage, true);
    }
```

Listing 3.8: Revised ETHPlatformV2::openPositionETH()

Status This issue has been fixed in this commit: `a146a46`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `CVI` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., `oracle` addition, fee adjustment, and parameter setting). It also has the privilege to regulate or govern the flow of assets among the involved components.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show representative privileged operations in the `CVI` protocol.

```
    function setFeesCollector(IFeesCollector _newCollector) external override onlyOwner
        {
        if (address(feesCollector) != address(0) && address(token) != address(0)) {
            token.approve(address(feesCollector), 0);
        }

        feesCollector = _newCollector;

        if (address(_newCollector) != address(0) && address(token) != address(0)) {
            token.approve(address(_newCollector), uint256(-1));
        }
    }

    function setRequestFeesCalculator(IRequestFeesCalculator _newRequestFeesCalculator)
        external override onlyOwner {
        requestFeesCalculator = _newRequestFeesCalculator;
    }

    function setCVIOracle(ICVIOracleV3 _newCVIOracle) external override onlyOwner {
        cviOracle = _newCVIOracle;
    }

    function setUniswapOracle(IUniswapOracle _newUniswapOracle) external override
        onlyOwner {
        uniswapOracle = _newUniswapOracle;
    }

    function setRebaseLag(uint8 _newRebaseLag) external override onlyOwner {
        rebaseLag = _newRebaseLag;
    }
```

Listing 3.9: Various Setters in CVI

We emphasize that the privilege assignment with various protocol contracts is necessary and required for proper protocol operations. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance will be managed by a multi-sig account.

We point out that a compromised `owner` account would allow the attacker to add a malicious setting to steal funds in current protocol, which directly undermines the assumption of the `CVI` protocol.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed and partially mitigated with a multi-sig account to regulate the governance privileges.

4 Conclusion

In this audit, we have analyzed the CVI design and implementation. The system presents a unique, robust offering as a decentralized non-custodial ecosystem that brings the sophisticated and very popular "market fear index" to the crypto market and is created by computing a decentralized volatility index from cryptocurrency option prices, together with analyzing the market's expectation of future volatility. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Furthermore, 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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

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

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

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

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