SMART CONTRACT AUDIT REPORT

for

PERPETUAL PROTOCOL

Prepared By: Shuxiao Wang

Hangzhou, China

Sep. 7, 2020

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	Sep. 7, 2020	Chiachih Wu	Final Release
1.0-rc1	Sep. 3, 2020	Chiachih Wu	Release Candidate #1

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 **Perpetual 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 branch of Perpetual Protocol 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 Perpetual Protocol

Perpetual Protocol, formerly known as Strike Protocol, is designed as a decentralized perpetual contract trading protocol for a list of assets with Uniswap-inspired Automated Market Makers (AMMs). It also has a built-in Liquidity Reserve which backs and secures the AMMs, and a build-in staking pool that provides a backstop for each virtual market. Similar to Uniswap, traders can trade with virtual AMMs without counter-parties, PERP token holders can stake PERPs to staking pool and collect transaction fees.

The basic information of Perpetual Protocol is as follows:

Table 1.1: Basic Information of Perpetual Protocol

Item	Description
Issuer	Perpetual Protocol
Website	https://perp.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Sep. 7, 2020

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

- https://github.com/Strike-Protocol/strike-monorepo.git (6136a33)

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.

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

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

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 Perpetual Protocol implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	1	■
Low	3	■■■
Informational	9	■■■■■■■■■
Total	14	

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, 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 9 informational recommendations.

Table 2.1: Key Perpetual Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility With Deflationary Tokens in ClearingHouse::addMargin()	Business Logics	Fixed
PVE-002	Info.	Redundant ERC20 Transfers in ClearingHouse::transferFee()	Business Logics	Fixed
PVE-003	Info.	Missed Events and Error Handling in InsuranceFund	Coding Practices	Fixed
PVE-004	Medium	Business Logic Error in InsuranceFund::getTokenWithMaxValue()	Business Logics	Fixed
PVE-005	Info.	Business Logic Error in RewardsDistribution::distributeRewards()	Business Logics	Fixed
PVE-006	Info.	Gas Optimization in RewardsDistribution::removeRewardsDistribution()	Business Logics	Fixed
PVE-007	Info.	Missed initializer Modifiers	Business Logics	Fixed
PVE-008	High	Reentrancy Risk in ClearingHouse::settlePosition()	Security Features	Fixed
PVE-009	Info.	Missed Sanity Checks Against _amm.open in ClearingHouse	Coding Practices	Fixed
PVE-010	Info.	Configurable Constant Variable in Amm	Business Logics	Confirmed
PVE-011	Info.	Better Handling of Ownership Transfers	Business Logics	Fixed
PVE-012	Low	Incompatibility With approve/transferFrom Race Prevention Tokens	Business Logics	Fixed
PVE-013	Info.	Wrong Comments in StakingReserve::getUnstakableBalance()	Business Logics	Fixed
PVE-014	Low	Business Logic Error in StakingReserve::getLockedBalance()	Business Logics	Fixed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incompatibility With Deflationary Tokens in ClearingHouse::addMargin()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: ClearingHouse.sol
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the ClearingHouse contract, the addMargin() external function allows an user to add margin to increase the margin ratio of her personal position. After updating the position stats, the addMargin() function requires the msg.sender to transfer in the _amm.quoteAsset() tokens with the transferFrom () handler. When transferring standard ERC20 tokens, these asset-transferring routines work as expected: namely the account's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts

```
    function addMargin(Amm _amm, Decimal.decimal calldata _addedMargin) external
        whenNotPaused() nonReentrant() {
        // update margin part in personal position
        address trader = _msgSender();
        updateMargin(_amm, trader, MixedDecimal.fromDecimal(_addedMargin));

        // transfer token from trader
        DecimalERC20.transferFrom(_amm.quoteAsset(), trader, address(clearingHouseVault)
            , _addedMargin);

        // emit event
        emit MarginAdded(trader, address(_amm), _addedMargin.toUint());
    }
```

Listing 3.1: ClearingHouse::addMargin()

However, in the cases of deflationary tokens, as shown in the above code snippets, the input amount may not be equal to the received amount due to the charged (and burned) transaction fee. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary tokens. Apparently, these balance inconsistencies are damaging to accurate portfolio management and affect protocol-wide operation and maintenance.

Recommendation Check the `_amm.quoteAsset()` balance before and after the `transferFrom()` call.

Status This issue has been addressed by checking the token balance before and after the `transferFrom()` call in this PR: 1204.

3.2 Redundant ERC20 Transfers in ClearingHouse::transferFee()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ClearingHouse.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the `ClearingHouse` contract, the internal function `transferFee()` is invoked by `openPositioin()` and `closePosition()` for collecting the fee from the trader. While reviewing the logic of fee transfers, we identified that the current implementation could be improved by reducing one `transferFrom()` call, which would reduce gas consumption.

```
    function transferFee(address _from, Amm _amm, Decimal.decimal memory _positionNotional)
        internal
        returns (Decimal.decimal memory)
    {
        (Decimal.decimal memory toll, Decimal.decimal memory spread) = ammMgr.calcFee(
            _amm, _positionNotional);
        if (toll.toUint() == 0 && spread.toUint() == 0) {
            return Decimal.zero();
        }

        IERC20 quoteAsset = _amm.quoteAsset();
        Decimal.decimal memory fee = toll.addD(spread);
        DecimalERC20.transferFrom(quoteAsset, _from, address(this), fee);
```

```
891         // transfer spread to insurance fund
892         DecimalERC20.transfer(quoteAsset, address(clearingHouseVault), spread);
893         clearingHouseVault.transferToInsuranceFund(quoteAsset, spread);

895         // transfer toll to ammMgr
896         DecimalERC20.transfer(quoteAsset, address(ammMgr), toll);
897         ammMgr.increaseToll(_amm, toll);

899         return fee;
900     }
```

Listing 3.2: ClearingHouse.sol

As shown in the code snippets above, the first `transferFrom()` moves `fee` of `quoteAsset` from `_from` to `address(this)` in line 889. Note that `fee` equals to `toll + spread` as shown in line 888. Later on, those `toll + spread` is `transfer()`'ed to `clearingHouseVault` and `ammMgr` separately in lines 892 and 896. Since both of the latter two `transfer()` calls consume `quoteAsset` tokens of `address(this)`, we could simplify them by transferring tokens from the `_from` address to `clearingHouseVault` and `ammMgr` directly. This essentially reduce the gas consumption of one `call`.

Recommendation Remove the first `transferFrom()` call and `transferFrom()` from `_from` to `clearingHouseVault` and `ammMgr` directly.

Status This issue has been addressed in this PR: 1195.

3.3 Missed Events and Error Handling in InsuranceFund

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `InsuranceFund.sol`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [3]

Description

In the `InsuranceFund` contract, the owner adds and removes the quote tokens through the `addToken()` and `removeToken()` functions with the `quoteTokens` array which keeps all the supported quote tokens. However, we noticed that the error handling logic is missed in these functions. As shown in the following code snippets, when we `addToken()` an existing quote token, the function simply returns with no error code. Also, when we `removeToken()` a quote token which has not been added yet, the function returns because of `idx == -1` in line 887 with no error code again.

```
    function addToken(IERC20 _token) external onlyOwner {
        if (!isQuoteTokenExisted(_token)) {
```

```
            quoteTokens.push(_token);
        }
    }

    function removeToken(IERC20 _token) external onlyOwner {
        int256 idx = getTokenIndex(_token);
        if (idx == -1) {
            return;
        }
```

Listing 3.3: InsuranceFund.sol

We suggest to revert the transaction when those error cases happen. Moreover, since adding/removing quote tokens are important updates in Perpetual Protocol, we suggest to emit events when a new quote token is added or an existing quote token is removed.

Recommendation Revert the transaction when adding/removing quote tokens fail and emit events when quote tokens are added/removed.

```
    function addToken(IERC20 _token) external onlyOwner {
        if (!isQuoteTokenExisted(_token)) {
            quoteTokens.push(_token);
            emit TokenAdded(address(_token), quoteTokens.length - 1);
        } else {
            revert("token existed");
        }
    }

    function removeToken(IERC20 _token) external onlyOwner {
        int256 idx = getTokenIndex(_token);
        require(idx >= 0, "token not existed");
        ...
        emit TokenRemoved(address(_token));
```

Listing 3.4: InsuranceFund.sol

Status This issue has been addressed in this PR: 1197.

3.4 Business Logic Error in InsuranceFund::getTokenWithMaxValue()

- ID: PVE-004
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `InsuranceFund.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the `InsuranceFund` contract, the owner adds and removes the quote tokens through the `addToken()` and `removeToken()` functions with the `quoteTokens` array which keeps all the supported quote tokens. While removing a quote token, the tokens withheld are exchanged to a `outputToken` as shown in the following code snippet. However, the business logic to choose the `outputToken` has some flaws such that the smallest quoteToken cannot be selected correctly.

```
        // exchange and transfer to the smallest quoteToken. if no more quoteToken, buy
            protocol tokens
        // TODO use curve or balancer fund token for pooling the fees will be less
            painful
        address outputToken = getTokenWithMaxValue(_token);
        if (outputToken == address(0)) {
            outputToken = address(perpToken);
        }
        swapInput(_token, IERC20(outputToken), balanceOf(_token), Decimal.zero());
```

Listing 3.5: InsuranceFund::removeToken()

As shown in the following code snippet, when `numOfQuoteTokens <= 1`, we have no choice but return the first quote token or the zero address which fails over to the `perpToken`. When `numOfQuoteTokens >= 2`, the for-loop in line 97 – 108 tends to quote each quote token based on the `_denominatedToken` and get the `maxValueToken`. However, the range of `i` in line 97 simply skip the first and the last quote token with no reason.

```
86    function getTokenWithMaxValue(IERC20 _denominatedToken) internal view returns (
          address) {
87        uint256 numOfQuoteTokens = quoteTokens.length;
88        if (numOfQuoteTokens == 0) {
89            return address(0);
90        }
91        if (numOfQuoteTokens == 1) {
92            return address(quoteTokens[0]);
93        }

95        IERC20 maxValueToken;
96        Decimal.decimal memory valueOfMaxValueToken;
97        for (uint256 i = 1; i < numOfQuoteTokens - 1; i++) {
98            IERC20 quoteToken = quoteTokens[i];
99            Decimal.decimal memory quoteTokenValue = exchange.getInputPrice(
100               quoteToken,
101               _denominatedToken,
102               balanceOf(quoteToken)
103           );
104           if (quoteTokenValue.cmp(valueOfMaxValueToken) > 0) {
105               maxValueToken = quoteToken;
106               valueOfMaxValueToken = quoteTokenValue;
107           }
108       }
```

```
        return address(maxValueToken);
    }
```

Listing 3.6: InsuranceFund.sol

Based on the current implementation, `getTokenWithMaxValue()` returns `address(0)` when `numOfQuoteTokens == 2`. If `numOfQuoteTokens > 2`, the `maxValueToken` in `quoteTokens[]` is returned but the `quoteTokens[0]` `quoteTokens[quoteTokens.length-1]` are skipped. We believe that this is not what the business logic was designed.

Recommendation Fix the range of `i` in the for-loop.

Status This issue has been addressed in this PR: 1203.

3.5 Business Logic Error in RewardsDistribution::distributeRewards()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `RewardsDistribution.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the `RewardsDistribution` contract, the `distributeRewards()` function distributes `perpToken` to the addresses in the `distributions[]` array. As shown in the code snippet below, `_amount` of `perpToken` tokens are distributed in the `for`-loop which iterates the `distributions[]` array.

Within the `for`-loop, we notice that some sanity checks are performed in line 81. However, the checks are not implemented correctly. In particular, `distributions[i].destination != address(0)` prevents the rewards from being sent to zero addresses. And `distributions[i].amount.toUint()!= 0` skips zero transfers. Both checks are reasonable.

```
69    function distributeRewards(Decimal.decimal memory _amount) override public {
70        require(_msgSender() == address(perpToken), "caller is not PerpToken");
71        require(
72            DecimalERC20.balanceOf(perpToken, address(this)).toUint() >= _amount.toUint
                  (),
73            "RewardsDistribution does not have enough PERP to distribute"
74        );

76        // Iterate the array of distributions sending the configured amounts
77        // the size of the distributions array will be controlled by owner (DAO)
78        // owner should be aware of not making this array too large
```

```
79          Decimal.decimal memory remainder = _amount;
80          for (uint256 i = 0; i < distributions.length; i++) {
81        if (distributions[i].destination != address(0) || distributions[i].amount.toUint() != 0) {
82                remainder = remainder.subD(distributions[i].amount);

84                // Transfer the PERP
85                DecimalERC20.transfer(perpToken, distributions[i].destination, distributions[i].amount);
```

Listing 3.7: RewardsDistribution.sol

The buggy part is that the two checks are `OR`'ed instead of `AND`'ed in line 81. It means the rewards could be distributed to a zero address if the amount is non-zero. Also, zero transfers are allowed if the destination address is not a zero address.

Recommendation Fix the if statement in line 81.

```
69      function distributeRewards(Decimal.decimal memory _amount) override public {
70          require(_msgSender() == address(perpToken), "caller is not PerpToken");
71          require(
72              DecimalERC20.balanceOf(perpToken, address(this)).toUint() >= _amount.toUint(),
73              "RewardsDistribution does not have enough PERP to distribute"
74          );

76          // Iterate the array of distributions sending the configured amounts
77          // the size of the distributions array will be controlled by owner (dao)
78          // owner should be aware of not making this array too large
79          Decimal.decimal memory remainder = _amount;
80          for (uint256 i = 0; i < distributions.length; i++) {
81        if (distributions[i].destination != address(0) && distributions[i].amount.toUint() != 0) {
82                remainder = remainder.subD(distributions[i].amount);

84                // Transfer the PERP
85                DecimalERC20.transfer(perpToken, distributions[i].destination, distributions[i].amount);
```

Listing 3.8: RewardsDistribution.sol

Status This issue has been addressed in this PR: 1195.

3.6 Gas Optimization in RewardsDistribution::removeRewardsDistribution()

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: RewardsDistribution.sol
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.5, the `distributions[]` array keeps the addresses and amounts for reward distribution. Meanwhile, it should be carefully maintained by the privileged user (i.e., the user who can pass the `onlyOwner` check). While reviewing the `removeRewardsDistribution()` function that removes the element indexed by `_index` from the `distributions[]` array, we notice that there's one trivial trick to reduce the gas consumption. Especially, when we have a huge `distributions[]` array, the improvement saves a lot of gas!

```
121     function removeRewardsDistribution(uint256 _index) external onlyOwner {
122         require(_index <= distributions.length - 1, "index out of bounds");

124         // shift distributions indexes across
125         for (uint256 i = _index; i < distributions.length - 1; i++) {
126             distributions[i] = distributions[i + 1];
127         }
128         distributions.pop();

130         // Since this function must shift all later entries down to fill the
131         // gap from the one it removed, it could in principle consume an
132         // unbounded amount of gas. However, the number of entries will
133         // presumably always be very low.
134     }
```

Listing 3.9: RewardsDistribution.sol

The trick is that we could simply replace the element to be removed with the last element in the array and `pop()` the last element out. This reduces a lot of gas if you need to walk through a huge array and replace each element with the next element as what the current implementation is (line 125 – 127).

Recommendation Replace the element to be removed with the last element and `pop()` the last element out.

```
    function removeRewardsDistribution(uint256 _index) external onlyOwner {
        require(_index <= distributions.length - 1, "index out of bounds");
```

```
        distributions[_index] = distributions[distributions.length - 1];
        distributions.pop();
    }
```

Listing 3.10: RewardsDistribution.sol

Status This issue has been addressed in this PR: 1195.

3.7 Missed initializer Modifiers

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: InflationMonitor.sol, SupplySchedule.sol
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the current prototype design, the `initialize()` function plays an important role to perform what the constructor should have done in an easier way. Since the `initialize()` function is typically `public`, a common practice is applying the `initializer` modifier on it, which is part of the `Initializable` contract provided by OpenZeppelin. With the help of the `initializer` modifier, bad actors have no chance to call the critical `initialize()` function again.

However, while reviewing all `initialize()` functions in Perpetual Protocol, we identified two cases that might have problems due to the missed `initializer` modifiers. As shown in the code snippets bellow, the `InflationMonitor` contract has a public `initialize()` (without the modifier) to initialize the `perpToken` and the `shutdownThreshold`, which we certainly do not expect a bad actor to set it again. The `SupplySchedule` contract also has a few important system parameters (initialized in the `initialize()` without the `initializer` modifier), which opens a big attack surface.

```
    function initialize(PerpToken _perpToken) public {
        __Ownable_init();

        perpToken = _perpToken;
        shutdownThreshold = Decimal.one().divScalar(10);
    }
```

Listing 3.11: InflationMonitor.sol

```
    function initialize(PerpToken _perpToken, uint256 _inflationRate, uint256 _decayRate
        , uint256 _mintDuration)
        public
    {
        __Ownable_init();
```

```
        perpToken = _perpToken;
        inflationRate = Decimal.decimal(_inflationRate);
        mintDuration = _mintDuration;
        decayRate = Decimal.decimal(_decayRate);
    }
```

Listing 3.12: SupplySchedule.sol

Fortunately, the `__Ownable_init()` function invoked by both `initialize()` functions has the `initializer` modifier applied, which prevents bad actors from re-entering the `initialize()` functions. We still suggest to explicitly use the `initializer` modifier on each `initialize()` function.

Recommendation Add the `initializer` modifier to the `initialize()` functions.

Status This issue has been addressed in this PR: 1197.

3.8 Reentrancy Risk in ClearingHouse::settlePosition()

- ID: PVE-008
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `ClearingHouse.sol`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [4]

Description

In the `ClearingHouse` contract, the `addMargin()` allows a `trader` to add margin to her position while the `settlePosition()` function allows her to settle the position. While reviewing the adding/settling mechanism, we identified a reentrancy risk such that a bad actor could `settlePosition()` a position but leave a batch of quote assets inside the `ClearingHouse` without book-keeping records. And, that batch of unknown quote asset is actually the margin `addMargin()`'ed into the position in the same transaction.

The problem is essentially caused by doing `settlePosition()` inside the `addMargin()` call due to the support of ERC777 (or similar tokens which support a callback mechanism) [16]. Specifically, as shown in the code snippet below, `addMargin()` performs `transferFrom()` (line 217) to collect the quote assets from the `trader` into the `ClearingVault`. If that quote asset is an ERC777 token, the bad actor could intercept the control flow before the asset flows into the `clearingHouseVault`.

```
211    function addMargin(Amm _amm, Decimal.decimal calldata _addedMargin) external
           whenNotPaused() nonReentrant() {
212        // update margin part in personal position
213        address trader = _msgSender();
214        updateMargin(_amm, trader, MixedDecimal.fromDecimal(_addedMargin));
```

```
        // transfer token from trader
        DecimalERC20.transferFrom(_amm.quoteAsset(), trader, address(clearingHouseVault)
            , _addedMargin);

        // emit event
        emit MarginAdded(trader, address(_amm), _addedMargin.toUint());
    }
```

Listing 3.13: ClearingHouse.sol

Now, `settlePosition()` deletes all the book-keeping records about the position which is `addMargin()`'ed earlier (line 258).

```
248     function settlePosition(Amm _amm) external {
249         // check condition
250         requireAmm(_amm);
251         require(!_amm.open(), "amm is open");

253         address trader = _msgSender();
254         Position memory pos = getPosition(_amm, trader);
255         requirePositionSize(pos.size);

257         // update position
258         deletePosition(_amm, trader);
```

Listing 3.14: ClearingHouse::settlePosition()

Later on, the `settledValue` amount of the quote assets is withdrawn by the `trader` (a.k.a. the bad actor) in line 282. When the code flow goes back to `addMargin()`'s `transferFrom()` call, the same amount of quote assets is transferred into the `ClearingHouseVault`. But, there's no record for that batch of assets.

```
279         // transfer token based on settledValue
280         if (settledValue.toUint() != 0) {
281             IERC20 quoteAsset = _amm.quoteAsset();
282             clearingHouseVault.withdraw(quoteAsset, trader, settledValue);
283         }
```

Listing 3.15: ClearingHouse::settlePosition()

Recommendation Add the necessary reentrancy guard to `settlePosition()`.

Status This issue has been addressed by adding reentrancy guard to `settlePosition()` in this PR: 1270.

3.9 Missed Sanity Checks Against _amm.open in ClearingHouse

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ClearingHouse.sol`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [3]

Description

While reviewing the `ClearingHouse` contract, we notice that the `_amm` variable is used for the interactions with a specific vAMM. Many of the use cases have sanity checks on the `_amm`, a few cases can still be improved.

Case I The `openPosition()` fails to ensure the `_amm` is in the `open` state (i.e., `_amm.open == true`). When the execution goes into `implOpenPosition()` and eventually invokes `_amm.swapInputWithMinBaseAsset()`, the transaction would be reverted due to the `onlyOpen()` modifier.

```
    function openPosition(
        Amm _amm,
        Side _side,
        Decimal.decimal calldata _quoteAssetAmount,
        Decimal.decimal calldata _leverage,
        Decimal.decimal calldata _minBaseAssetAmount
    ) external whenNotPaused() nonReentrant() {
        // check conditions
        requireAmm(_amm);
        requireNonZeroInput(_quoteAssetAmount);
        requireNonZeroInput(_leverage);
        requireEnoughMarginRatio(MixedDecimal.fromDecimal(Decimal.one()).divD(_leverage)
            );

        // update position
        address trader = _msgSender();
        PositionResp memory positionResp = implOpenPosition(
            PositionArgs(_amm, trader, _side, _quoteAssetAmount.mulD(_leverage),
                _leverage, _minBaseAssetAmount)
        );
```

Listing 3.16: ClearingHouse.sol

```
    function swapInputWithMinBaseAsset(
        Dir _dir,
        Decimal.decimal calldata _quoteAssetAmount,
        Decimal.decimal calldata _minValueOfBaseAssetAmount
```

```
    ) external onlyOpen onlyCounterParty returns (Decimal.decimal memory) {
```

Listing 3.17: Amm.sol

Case II Similar to the previous case, both `closePosition()` and `liquidate()` hit the `onlyOpen()` function `_amm.forceSwapOutput()` without checking `_amm.open()` in the first place, which would be a waste of gas.

```
    function closePosition(Amm _amm) external whenNotPaused() nonReentrant() {
        // check conditions
        requireAmm(_amm);

        // update position
        address trader = _msgSender();
        PositionResp memory positionResp = internalClosePosition(_amm, trader);
```

Listing 3.18: ClearingHouse.sol

```
    function liquidate(Amm _amm, address _trader) external nonReentrant() {
        // check conditions
        requireAmm(_amm);
        require(
            getMarginRatio(_amm, _trader).subD(maintenanceMarginRatio).toInt() < 0,
            "Margin ratio is larger than min requirement"
        );
        address liquidator = _msgSender();
        Position memory liquidatorPos = getUnadjustedPosition(_amm, liquidator);
        // TODO have a short message
        require(liquidatorPos.blockNumber != _blockNumber(), "can't open and liquidate
            in the same block");

        // update position
        PositionResp memory positionResp = internalClosePosition(_amm, _trader);
```

Listing 3.19: ClearingHouse.sol

```
    function forceSwapOutput(Dir _dir, Decimal.decimal calldata _baseAssetAmount)
        external
        onlyOpen
        onlyCounterParty
        returns (Decimal.decimal memory)
    {
        return implSwapOutput(_dir, _baseAssetAmount, true);
    }
```

Listing 3.20: Amm.sol

Case III The same improvement is also applicable to `payFunding()`. When it reaches `_amm.settleFunding()`, the unchecked `_amm.open` reverts the transaction which could have been reverted in the first line of `payFunding()`.

```
    function payFunding(Amm _amm) external {
        requireAmm(_amm);

        // must copy the baseAssetDeltaThisFundingPeriod
        SignedDecimal.signedDecimal memory baseAssetDeltaThisFundingPeriod =
            SignedDecimal.signedDecimal(
            _amm.baseAssetDeltaThisFundingPeriod()
        );

        SignedDecimal.signedDecimal memory premiumFraction = _amm.settleFunding();
```

Listing 3.21: ClearingHouse.sol

```
236     function settleFunding() external onlyOpen returns (SignedDecimal.signedDecimal
            memory) {
237         require(_blockTimestamp() >= nextFundingTime, "settle funding too early");

239         // premium = twapMarketPrice - twapIndexPrice
240         // timeFraction = fundingPeriod(1 hour) / 1 day
241         // premiumFraction = premium * timeFraction
```

Listing 3.22: Amm.sol

Recommendation Ensure `_amm.open == true` in the beginning of `ClearingHouse` function.

Status This issue has been addressed by refactoring the `requireAmm()` function with the `_open` parameter to validate the state of the vAMM in this PR: 1270.

3.10 Configurable Constant Variable in Amm

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Amm.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

The `fundingPeriod` is an important variable for the `Amm` to update the funding rate. As shown in the code snippet below, the `settleFunding()` external function allows anyone to update the funding rate if the current block timestamp is beyond the planned time (i.e., `nextFundingTime`). The new funding rate is calculated based on the `premium` and the `timeFraction` as indicated in the comments (lines 239 – 241). Here, we notice that the `timeFraction` is derived by a number of seconds (`fundingPeriod`) divided by a constant, 86,400 (`1 days`). The idea is that we want to gradually kick in the difference between `twapMarketPrice` and `twapIndexPrice`. However, the `fundingPeriod` is not a constant due to the fact that the `initialize()` function can set it as any given value. This may lead to unexpected side effects. For example, if it's greater than `1 days`, the `premiumFraction` would be greater than the `premium` itself.

```
    function settleFunding() external onlyOpen returns (SignedDecimal.signedDecimal
        memory) {
        require(_blockTimestamp() >= nextFundingTime, "settle funding too early");

        // premium = twapMarketPrice - twapIndexPrice
        // timeFraction = fundingPeriod(1 hour) / 1 day
        // premiumFraction = premium * timeFraction
        Decimal.decimal memory underlyingPrice = getUnderlyingPrice();
        SignedDecimal.signedDecimal memory premium = MixedDecimal.fromDecimal(
            getTwapPrice(spotPriceTwapInterval)).subD(
            underlyingPrice
        );
        SignedDecimal.signedDecimal memory premiumFraction = premium.mulScalar(
            fundingPeriod).divScalar(int256(1 days));

        // update funding rate = premiumFraction / twapIndexPrice
        updateFundingRate(premiumFraction.divD(underlyingPrice));
```

Listing 3.23: Amm.sol

```
        quoteAssetReserve = Decimal.decimal(_quoteAssetReserve);
        baseAssetReserve = Decimal.decimal(_baseAssetReserve);
        tradeLimitRatio = Decimal.decimal(_tradeLimitRatio);
        fluctuation = Decimal.decimal(_fluctuation);
        fundingPeriod = _fundingPeriod;
        fundingBufferPeriod = _fundingPeriod.div(2);
```

Listing 3.24: Amm.sol:: initialize ()

Recommendation Set `fundingPeriod` as a constant variable (i.e., 3600).

Status After discussing with the team, they confirmed this issue and decided to leave as is.

3.11 Better Handling of Privilege Transfers

- ID: PVE-011
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Targets: Ownable.sol
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

Perpetual Protocol implements a rather basic access control mechanism that allows a privileged account, i.e., _owner, to be granted exclusive access to typically sensitive functions (e.g., adding/removing a quote token). Because of the privileged access and the implications of these sensitive functions, the _owner account is essential for the protocol-level safety and operation. In the following, we elaborate with the _owner account.

Within the governing contract Ownable, a specific function, i.e., transferOwnership(address newOwner), is provided to allow for possible _owner updates. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the newOwner parameter is always correctly provided. However, in the unlikely situation, when an incorrect newOwner is provided, the contract ownership may be lost forever, which would be devastating for Perpetual Protocol operation and maintenance.

```
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
```

Listing 3.25: Ownabe.sol

As a common best practice, instead of achieving the owner update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the owner update intent and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract ownership to an uncontrolled address. In other words, this two-step procedure ensures that an owner public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

Recommendation Implement a two-step approach for owner update (or transfer): setOwner() and acceptOwner().

```
    function setOwner(address newOwner) public onlyOwner {
        require(newOwner != address(0), "Ownable: zero address");
```

```
        require(newOwner != _owner, "Ownable: same as original");
        require(newOwner != _candidate, "Ownable: same as candidate");
        _candidate = newOwner;
    }

    function updateOwner() public {
        require(_candidate != address(0), "Ownable: candidate is zero address");
        require(_candidate == _msgSender(), "Ownable: not the new owner");

        emit OwnershipTransferred(_owner, _candidate);
        _owner = _candidate;
        _candidate = address(0);
    }
```

Listing 3.26: Ownabe.sol

Status This issue has been addressed by implementing the two-step approach in this PR: 1266.

3.12 Incompatibility With approve/transferFrom Race Prevention Tokens

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Targets: `InsuranceFund.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In some cases, the `InsuranceFund` contract needs to deal with exchange services for swapping a token to another. For example, while removing a quote token, the token balance withheld would be exchanged to an existing quote token or the perp token as implemented in the `removeToken()` function. The underlying internal function to swap the tokens (e.g., `swapInput()`) needs to `approve()` the exchange to spend certain amount of the token balance of the `InsuranceFund` contract, which is a common practice of DEXs.

```
    function swapInput(
        IERC20 inputToken,
        IERC20 outputToken,
        Decimal.decimal memory inputTokenSold,
        Decimal.decimal memory minOutputTokenBought
    ) internal returns (Decimal.decimal memory received) {
        if (inputTokenSold.toUint() == 0) {
            return Decimal.zero();
        }
```

```
165         DecimalERC20.approve(inputToken, address(exchange), inputTokenSold);
166         received = exchange.swapInput(inputToken, outputToken, inputTokenSold,
                minOutputTokenBought);
167         require(received.toUint() > 0, "Exchange swap error");
168     }
```

Listing 3.27: InsuranceFund.sol

However, for dealing with the approve/transferFrom race condition issue [2], many ERC20 tokens implement their `approve()` handlers as follows:

```
    function approve(address guy, uint wad) public stoppable returns (bool) {
    require(_approvals[msg.sender][guy] == 0 || wad == 0); //take care of re-approve.
        return super.approve(guy, wad);
    }
```

Listing 3.28: approve/transferFrom Race Prevention Token

As a result, if the current implementation of `swapInput()` is about to exchange such tokens, the `approve()` call in line 165 simply reverts. We suggest to always `approve(0)` before `approve()`'ing the real amount to facilitate different `approve()` implementations.

Recommendation Set the approval to 0 before the real `approve()` call.

Status This issue has been addressed by refactoring the `DecimalERC20` utils in this PR: 1263.

3.13 Wrong Comments in StakingReserve::getUnstakableBalance()

- ID: PVE-013
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Targets: `StakingReserve.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

The `getUnstakableBalance()` retrieves the `lockedBalance` at the one after the next epoch as shown in the code below. However, the comments in line 366 suggests the `lockedBalance` at the next epoch which is inconsistent to the implementation.

```
366     // unstakable = lockedBalance@NextEpoch
367     function getUnstakableBalance(address _staker) public view returns (Decimal.decimal
            memory) {
368         return getLockedBalance(_staker, nextEpochIndex().add(1)).locked;
```

```
    }
```

Listing 3.29: StakingReserve.sol

Recommendation Fix the comments as follows:

```
    // unstakable = lockedBalance@NextEpoch+1
    function getUnstakableBalance(address _staker) public view returns (Decimal.decimal
        memory) {
        return getLockedBalance(_staker, nextEpochIndex().add(1)).locked;
    }
```

Listing 3.30: StakingReserve.sol

Status This issue has been addressed by revising the comments in this PR: 1260.

3.14 Business Logic Error in StakingReserve::getLockedBalance()

- ID: PVE-014
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Targets: `StakingReserve.sol`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the `StakingReserve` contract, stakers can use the `stake()` and `unstake()` functions to increase or decrease the staking. Specifically, the `stakeBalance.lockedBalanceMap[]` array keeps the locked amount and the time-weighted locked amount for each staker in different epoch. While reviewing the staking mechanism, we identified a business logic error in the public view function `getLockedBalance()`.

As shown in the following code snippet, the `getLockedBalance()` function walks through the `stakeBalance.lockedBalanceMap[]` array to find out the closest previous `lockedBalance`. However, in line 378, the index cannot reach 0, which means if the target `lockedBalance` is at `stakeBalance.lockedBalanceMap[0]`, there's no way to find it out. As a result, zero locked amount and zero time-weighted locked amount would be returned in line 384.

```
373     function getLockedBalance(address _staker, uint256 _epochIndex) public view returns
            (LockedBalance memory) {
374         StakeBalance storage stakeBalance = stakeBalanceMap[_staker];
375         if (0 == _epochIndex) {
376             return stakeBalance.lockedBalanceMap[_epochIndex];
377         }
378         for (uint256 i = _epochIndex; i > 0; i--) {
```

```
            LockedBalance memory lockedBalance = stakeBalance.lockedBalanceMap[i];
            if (lockedBalance.exist) {
                return lockedBalance;
            }
        }
        return LockedBalance(false, Decimal.zero(), Decimal.zero());
    }
```

Listing 3.31: StakingReserve.sol

If a staker is about to `unstake()` some assets but the closest existing `lockedBalance` is at index 0 (as mentioned above), her assets would be locked eternally.

Recommendation Take care the `stakeBalance.lockedBalanceMap[0]` case.

```
    function getLockedBalance(address _staker, uint256 _epochIndex) public view returns
        (LockedBalance memory) {
        StakeBalance storage stakeBalance = stakeBalanceMap[_staker];
        if (0 == _epochIndex) {
            return stakeBalance.lockedBalanceMap[_epochIndex];
        }
        for (uint256 i = _epochIndex; i > 0; i--) {
            LockedBalance memory lockedBalance = stakeBalance.lockedBalanceMap[i];
            if (lockedBalance.exist) {
                return lockedBalance;
            }
        }
        if ( i == 0 ) {
            LockedBalance memory lockedBalance = stakeBalance.lockedBalanceMap[i];
            if (lockedBalance.exist) {
                return lockedBalance;
            }
        }
        return LockedBalance(false, Decimal.zero(), Decimal.zero());
    }
```

Listing 3.32: StakingReserve.sol

Status This issue has been addressed by traversing the `stakeBalance.lockedBalanceMap[]` until `i=0` in this PR: 1260.

3.15 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.0;` instead of `pragma solidity >=0.6.0;`.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, 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 need to kick in at the very moment when the contracts are being deployed in mainnet.

4 Conclusion

In this audit, we thoroughly analyzed the Perpetual Protocol design and implementation. The system presents a unique offering of perpetual contract trading of various digital assets and we are impressed by the design and implementation. 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, 17].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [18] 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] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

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

[4] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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 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. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://blog.peckshield.com/2020/04/19/erc777.

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

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