SMART CONTRACT AUDIT REPORT

for

PERPETUAL PROTOCOL

Prepared By: Shuxiao Wang

Hangzhou, China

Dec. 1, 2020

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	Dec. 1, 2020	Chiachih Wu	Final Release
1.0-rc	Nov. 23, 2020	Chiachih Wu	Release Candidate

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** contract 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 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	Dec. 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/Strike-Protocol/strike-monorepo.git (6136a33)

Also, the fixes resulted from this audit have been committed into this repo:

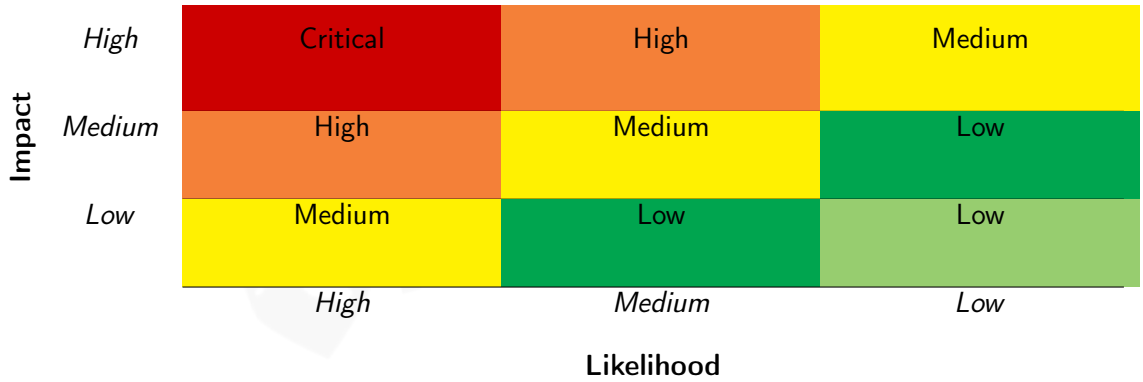
- https://github.com/perpetual-protocol/perp-contract

For example, commit 7715630 mentioned in Section 3 can be found here: https://github.com/perpetual-protocol/perp-contract/commit/7715630c8c561f10634fb0090a9f416f0ee41964

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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [7]:

- 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 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) [6], 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 1.3: 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

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 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	1	■
Informational	4	■■■■
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, 1 medium-severity vulnerability, 1 low-severity vulnerability, and 4 informational recommendations.

Table 2.1: Key Perpetual Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Signature Forgery Risk in executeMetaTransaction()	Business Logic	Fixed
PVE-002	Info.	Missed Ether Relaying in executeMetaTransaction()	Business Logic	Fixed
PVE-003	High	Confused Deputy in updatePriceFeed()	Coding Practices	Fixed
PVE-004	Info.	Missed Sanity Checks in erc20Transfer()	Business Logic	Fixed
PVE-005	Info.	Missed Sanity Checks in addAggregator()	Business Logic	Fixed
PVE-006	Info.	Gas Optimizations	Business Logic	Fixed
PVE-007	Low	Integer Underflow Risks in L2PriceFeed	Coding Practices	Fixed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Signature Forgery Risk in executeMetaTransaction()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `ClearingHouse.sol`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In Perpetual Protocol, the `MetaTxGateway` contract allows users to cross-layer call a whitelisted `to` address with arbitrary `functionSignature` data and a valid signature for verifying that the call is signed by the `from` address. As shown in the code snippet below, lines 130 – 131 `(metaTx, sigR, sigS, sigV)` tuple is verified by the `verify()` function.

```
112     function executeMetaTransaction(
113         address from,
114         address to,
115         bytes calldata functionSignature,
116         bytes32 sigR,
117         bytes32 sigS,
118         uint8 sigV
119     ) external payable returns (bytes memory) {
120         require(isInWhitelists(to), "!whitelisted");

122         MetaTransaction memory metaTx = MetaTransaction({
123             nonce: nonces[from],
124             from: from,
125             to: to,
126             functionSignature: functionSignature
127         });

129         require(
130             verify(from, domainSeperatorL1, metaTx, sigR, sigS, sigV)
131                 verify(from, domainSeperatorL2, metaTx, sigR, sigS, sigV),
132             "Meta tx Signer and signature do not match"
```

```
133         );

135         nonces[from] = nonces[from].add(1);
136         // Append userAddress at the end to extract it from calling context
137         (bool success, bytes memory returnData) = address(to).call(abi.encodePacked(
                functionSignature, from));

139         require(success, "executeMetaTx function call failed");
140         emit MetaTransactionExecuted(from, to, msg.sender, functionSignature);
141         return returnData;
142     }
```

Listing 3.1: MetaTxGateway.sol

However, as we review the `verify()` function implementation, we identify that the return value of `ecrecover()` is not properly checked, which leads to signature forgery risks. Specifically, the `ecrecover()` returns 0 when the signature is invalid. It means if the `from` address is `address(0)`, `signer == user` would always be `true`.

```
190     function verify(
191         address user,
192         bytes32 domainSeperator,
193         MetaTransaction memory metaTx,
194         bytes32 sigR,
195         bytes32 sigS,
196         uint8 sigV
197     ) internal view returns (bool) {
198         address signer = ecrecover(toTypedMessageHash(domainSeperator,
                hashMetaTransaction(metaTx)), sigV, sigR, sigS);
199         return signer == user;
200     }
```

Listing 3.2: MetaTxGateway.sol

By exploiting this vulnerability, a bad actor could generate arbitrary transactions signed by `address(0)` on the other layer. Since `address(0)` is typically used as a non-existing address (e.g., a blackhole address keeping the burned tokens), this could lead to critical damages.

Recommendation Ensure `signer != address(0)` in `verify()`.

Status This issue has been fixed in this commit: 7715630.

3.2 Missed Ether Relaying in MetaTxGateway::executeMetaTransaction()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MetaTxGateway.sol`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.1, the `executeMetaTransaction()` allows users to perform cross-layer function calls. While reviewing the implementation, we notice that the `executeMetaTransaction()` function is declared as a `payable` function. However, in line 137, the received Ethers are not sent to the `to` address (i.e., with `value()`).

```
112     function executeMetaTransaction(
113         address from,
114         address to,
115         bytes calldata functionSignature,
116         bytes32 sigR,
117         bytes32 sigS,
118         uint8 sigV
119     ) external payable returns (bytes memory) {
120         require(isInWhitelists(to), "!whitelisted");

122         MetaTransaction memory metaTx = MetaTransaction({
123             nonce: nonces[from],
124             from: from,
125             to: to,
126             functionSignature: functionSignature
127         });

129         require(
130             verify(from, domainSeperatorL1, metaTx, sigR, sigS, sigV)
131                 verify(from, domainSeperatorL2, metaTx, sigR, sigS, sigV),
132             "Meta tx Signer and signature do not match"
133         );

135         nonces[from] = nonces[from].add(1);
136         // Append userAddress at the end to extract it from calling context
137         (bool success, bytes memory returnData) = address(to).call(abi.encodePacked(
                functionSignature, from));

139         require(success, "executeMetaTx function call failed");
140         emit MetaTransactionExecuted(from, to, msg.sender, functionSignature);
141         return returnData;
```

```
142     }
```

Listing 3.3: MetaTxGateway.sol

Recommendation Relay the Ether to the `to` address.

Status The `payable` modifier is removed from `executeMetaTransaction()` in this commit: 2a54bd7.

3.3 Confused Deputy in RootBridge::updatePriceFeed()

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `InsuranceFund.sol`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [2]

Description

In the `RootBridge` contract, the `updatePriceFeed()` function allows the `priceFeed` contract to bridge the price data to layer 2. As shown in the code below, line 39 ensures the caller is the whitelisted `priceFeed` address. Later on, the `_price`, `_timestamp`, and other information are packed into `data`. In line 49, `callBridge()` is invoked to relay the `data` to the layer 2 price feed address, `_priceFeedAddrOnL2`.

```
32      function updatePriceFeed(
33          address _priceFeedAddrOnL2,
34          bytes32 _priceFeedKey,
35          Decimal.decimal calldata _price,
36          uint256 _timestamp,
37          uint256 _roundId
38      ) external returns (bytes32 messageId) {
39          require(address(priceFeed) == _msgSender(), "!priceFeed");

41          bytes4 methodSelector = IPriceFeed.setLatestData.selector;
42          bytes memory data = abi.encodeWithSelector(
43              methodSelector,
44              _priceFeedKey,
45              _price.toUint(),
46              _timestamp,
47              _roundId
48          );
49          return callBridge(_priceFeedAddrOnL2, data, DEFAULT_GAS_LIMIT);
50      }
```

Listing 3.4: RootBridge.sol

However, we notice that there's an external function `callOtherSideFunction()` in the `BaseBridge` contract which also invokes `callBridge()`. Since `callOtherSideFunction()` allows any caller to send in arbitrary `_contractOnOtherSide` and `_data` for some reason, this function could be abused by bad actors to perform `updatePriceFeed()` from non-whitelisted addresses. This leads to critical price manipulation attacks.

```
65      function callOtherSideFunction(address _contractOnOtherSide, bytes calldata _data,
            uint256 _gasLimit) external override returns (bytes32 messageId) {
66          return callBridge(_contractOnOtherSide, _data, _gasLimit);
67      }
```

Listing 3.5: BaseBridge.sol

Recommendation Whitelist the caller, callee, and function signature in the `callOtherSideFunction()`.

Status This issue has been addressed by removing the `callOtherSideFunction()` function in this commit: b3c8ccf.

3.4 Missed Sanity Check in BaseBridge::erc20Transfer()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BaseBridge.sol`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In Perpetual Protocol, the `BaseBridge` contract provides the `erc20Transfer()` function to relay ERC20 transfers between layer 1 and layer 2. As shown in the code snippets below, the `multiTokenTransfer()` internal function is invoked to collect ERC20 tokens into the `BaseBridge` contract, for example, in layer 1. Later on, the `multiTokenMediator` contract is called to relay the ERC20 transfer to the `_receiver` at layer 2. While reviewing the implementation, we notice that zero amount transfers are allowed in `erc20Transfer()`. However, as relaying ERC20 transfers consumes lot of gas including contract calls, it is not worth to relay zero transfers.

```
60      function erc20Transfer(IERC20 _token, address _receiver, Decimal.decimal calldata
            _amount) external override {
61          multiTokenTransfer(_token, _receiver, _amount);
62      }
```

Listing 3.6: BaseBridge.sol

```
72      function multiTokenTransfer(
73          IERC20 _token,
74          address _receiver,
75          Decimal.decimal memory _amount
76      ) internal virtual {
77          require(_receiver != address(0), "receiver is empty");
78          // transfer tokens from msg sender
79          _transferFrom(_token, _msgSender(), address(this), _amount);
81          // approve to multi token mediator and call 'relayTokens'
82          approveToMediator(_token);
83          // solhint-disable avoid-low-level-calls
84          (bool ret, ) = multiTokenMediator.call(
85              abi.encodeWithSelector(RELAY_TOKENS, address(_token), _receiver, _toUint(
                    _token, _amount))
86          );
87          require(ret, "BaseBridge: call relayTokens error");
88          emit Relayed(address(_token), _receiver, _amount.toUint());
89      }
```

Listing 3.7: BaseBridge.sol

Recommendation Ensure `_amount` is greater than 0 before relaying the token transfer.

```
60      function erc20Transfer(IERC20 _token, address _receiver, Decimal.decimal calldata
            _amount) external override {
61          if ( _amount.toUint() > 0 ) {
62              multiTokenTransfer(_token, _receiver, _amount);
63          }
64      }
```

Listing 3.8: BaseBridge.sol

Status This issue has been addressed by requiring `_amount.toUint()> 0` in this commit: d6735d9.

3.5 Missed Sanity Checks in addAggregator()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ChainlinkL1.sol`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In Perpetual Protocol, the `ChainlinkL1` contract provides the mechanism for gathering price data. In particular, the `updateLatestRoundData()` function allows an `aggregator` to post the latest price with a

valid _priceFeedKey which is associated with a non-zero priceFeedMap[_priceFeedKey]. For maintaining the priceFeedMap[] mapping, addAggregator() and removeAggregator() allow the owner to add/remove a (_priceFeedKey, _aggregator) pair.

```
63      function addAggregator(bytes32 _priceFeedKey, address _aggregator) external
            onlyOwner {
64          if (address(priceFeedMap[_priceFeedKey]) == address(0)) {
65              priceFeedKeys.push(_priceFeedKey);
66          }
67          priceFeedMap[_priceFeedKey] = AggregatorV3Interface(_aggregator);
68      }
```

Listing 3.9: ChainlinkL1.sol

However, the addAggregator() function fails o check if the newly added _aggregator is a non-zero address such that an invalid (_priceFeedKey, _aggregator) pair could be added (by mistake).

Recommendation Ensure address(_aggregator)!= address(0) in addAggregator().

```
63      function addAggregator(bytes32 _priceFeedKey, address _aggregator) external
            onlyOwner {
64          require(address(_aggregator) != address(0));
65          if (address(priceFeedMap[_priceFeedKey]) == address(0)) {
66              priceFeedKeys.push(_priceFeedKey);
67          }
68          priceFeedMap[_priceFeedKey] = AggregatorV3Interface(_aggregator);
69      }
```

Listing 3.10: ChainlinkL1.sol

Status This issue has been addressed by adding sanity checks in this commit: 5174355.

3.6 Gas Optimizations

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: RewardsDistribution.sol
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In Solidity, one common practice of removing an entry from an array is swapping the last entry of an array with the one to be removed and pop() the last entry. We do see the cases that use the trick to reduce gas consumption. However, there's one possible way to further reduce storage access.

```
70      function removeAggregator(bytes32 _priceFeedKey) external onlyOwner {
71          requireAggregatorExisted(getAggregator(_priceFeedKey));
72          delete priceFeedMap[_priceFeedKey];
73
74          uint256 length = priceFeedKeys.length;
75          for (uint256 i; i < length; i++) {
76              if (priceFeedKeys[i] == _priceFeedKey) {
77                  priceFeedKeys[i] = priceFeedKeys[length - 1];
78                  priceFeedKeys.pop();
79                  break;
80              }
81          }
82      }
```

Listing 3.11: ChainlinkL1::removeAggregator()

As shown in the code snippet above, the for-loop in lines 75 – 81 searches for the `_priceFeedKey` from the `priceFeedKeys[]` array, replace it with the last entry of `priceFeedKeys[]`, and `pop()` the last entry. As a corner case, when the entry to be removed is exactly the last entry, we could simply `pop()` it without extra `SSTORE` operations. Same logic applies to following cases:

```
66      function removeAggregator(bytes32 _priceFeedKey) external onlyOwner {
67          requireKeyExisted(_priceFeedKey, true);
68          delete priceFeedMap[_priceFeedKey];
69
70          uint256 length = priceFeedKeys.length;
71          for (uint256 i; i < length; i++) {
72              if (priceFeedKeys[i] == _priceFeedKey) {
73                  priceFeedKeys[i] = priceFeedKeys[length - 1];
74                  priceFeedKeys.pop();
75                  break;
76              }
77          }
78      }
```

Listing 3.12: L2PriceFeed::removeAggregator()

```
114     function removeRewardsDistribution(uint256 _index) external onlyOwner {
115         require(distributions.length != 0 && _index <= distributions.length - 1, "index
                out of bounds");
116
117         distributions[_index] = distributions[distributions.length - 1];
118         distributions.pop();
119     }
```

Listing 3.13: RewardsDistribution::removeRewardsDistribution()

```
114     function removeToken(IERC20 _token) external onlyOwner {
115         require(isQuoteTokenExisted(_token), "token not existed");
116
117         quoteTokenMap[address(_token)] = false;
118         uint256 quoteTokensLength = getQuoteTokenLength();
```

```
119         for (uint256 i = 0; i < quoteTokensLength; i++) {
120             if (quoteTokens[i] == _token) {
121                 quoteTokens[i] = quoteTokens[quoteTokensLength - 1];
122                 quoteTokens.pop();
123                 break;
124             }
125         }
```

Listing 3.14: InsuranceFund::removeToken()

Recommendation Check before swapping the array entries; `pop()` the array entry directly if the one to be removed is the last entry.

```
70     function removeAggregator(bytes32 _priceFeedKey) external onlyOwner {
71         requireAggregatorExisted(getAggregator(_priceFeedKey));
72         delete priceFeedMap[_priceFeedKey];
73
74         uint256 length = priceFeedKeys.length;
75         for (uint256 i; i < length; i++) {
76             if (priceFeedKeys[i] == _priceFeedKey) {
77                 if ( i != length-1 ) {
78                     priceFeedKeys[i] = priceFeedKeys[length - 1];
79                 }
80                 priceFeedKeys.pop();
81                 break;
82             }
83         }
84     }
```

Listing 3.15: ChainlinkL1::removeAggregator()

Status This issue has been addressed in these commits: 5174355 and 11e4a43.

3.7 Integer Underflow Risks in L2PriceFeed

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `L2PriceFeed.sol`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [2]

Description

In Perpetual Protocol, the `L2PriceFeed` contract relays the price feeds from layer 1 and provide view functions for layer 2 components to get the price data. While reviewing the implementation of those view functions, we identify that the price data provided could be invalid (i.e., always 0) in some extreme cases. As shown in the code snippet below, the `getPreviousPrice()` returns the `_numOfRoundBack`

price by indexing the `priceFeedMap[_priceFeedKey].priceData[]` array with `(len-_numOfRoundBack-1)` in line 194. To prevent out-of-bound access, line 193 ensures `_numOfRoundBack <= len`. However, the corner case, `_numOfRoundBack == len`, is missed.

```
189     function getPreviousPrice(bytes32 _priceFeedKey, uint256 _numOfRoundBack) public
            override view returns (uint256) {
190         require(isExistedKey(_priceFeedKey), "key not existed");

192         uint len = getPriceFeedLength(_priceFeedKey);
193         require(len > 0 && _numOfRoundBack <= len, "Not enough history");
194         return priceFeedMap[_priceFeedKey].priceData[len - _numOfRoundBack - 1].price;
195     }
```

Listing 3.16: L2PriceFeed.sol

If the caller passes in the `_numOfRoundBack` which is equal to `len`, `priceFeedMap[_priceFeedKey].priceData[-1].price` would be returned. In most cases, the price would be 0, which leads to abnormal trading behavior.

```
194     function getPreviousTimestamp(bytes32 _priceFeedKey, uint256 _numOfRoundBack) public
            override view returns (uint256) {
195         require(isExistedKey(_priceFeedKey), "key not existed");

197         uint256 len = getPriceFeedLength(_priceFeedKey);
198         require(len > 0 && _numOfRoundBack <= len, "Not enough history");
199         return priceFeedMap[_priceFeedKey].priceData[len - _numOfRoundBack - 1].
                timestamp;
200     }
```

Listing 3.17: L2PriceFeed.sol

Similar issue is identified in the `getPreviousTimestamp()` function in line 199.

Recommendation Ensure `_numOfRoundBack < len` as follows:

```
189     function getPreviousPrice(bytes32 _priceFeedKey, uint256 _numOfRoundBack) public
            override view returns (uint256) {
190         require(isExistedKey(_priceFeedKey), "key not existed");

192         uint len = getPriceFeedLength(_priceFeedKey);
193         require(len > 0 && _numOfRoundBack < len, "Not enough history");
194         return priceFeedMap[_priceFeedKey].priceData[len - _numOfRoundBack - 1].price;
195     }
```

Listing 3.18: L2PriceFeed.sol

```
194     function getPreviousTimestamp(bytes32 _priceFeedKey, uint256 _numOfRoundBack) public
            override view returns (uint256) {
195         require(isExistedKey(_priceFeedKey), "key not existed");

197         uint256 len = getPriceFeedLength(_priceFeedKey);
198         require(len > 0 && _numOfRoundBack < len, "Not enough history");
```

```
199         return priceFeedMap[_priceFeedKey].priceData[len - _numOfRoundBack - 1].
                timestamp;
200     }
```

Listing 3.19: L2PriceFeed.sol

Status This issue has been addressed in this commit: ea98cb9.

3.8 Other Suggestions

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 [8, 9, 10, 11, 13].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [14] 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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

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

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

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

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

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