SMART CONTRACT AUDIT REPORT

for

PERPETUAL PROTOCOL

Prepared By: Shuxiao Wang

PeckShield

Jan. 27, 2021

Document Properties

Client	Perpetual Protocol
Title	Smart Contract Audit Report
Target	Staking, Rewards Vesting, and Keeper
Version	1.0-rc1
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-rc1	Jan. 27, 2021	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 design document and related smart contract source code of the Perpetual Protocol's **Staking, Rewards Vesting, and Keeper** functionality, 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 Staking, Rewards Vesting, and Keeper contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About 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	Jan. 27, 2021

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

- https://github.com/perpetual-protocol/perp-contract (5247397)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ 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 the OWASP Risk Rating Methodology [6]:

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

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

Table 1.3: The Full List of Check Items

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business 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 given source code of the Perpetual Protocol's **Staking, Rewards Vesting, and Keeper** functionality. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	1	■
Informational	3	■ ■ ■
Total	4	

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 low-severity vulnerability, and 3 informational recommendations.

Table 2.1: Key Staking, Rewards Vesting, and Keeper Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Missed Access Control in KeeperReward L1/L2	Business Logic	Confirmed
PVE-002	Info.	Unused Mapping in PerpRewardVesting	Business Logic	Fixed
PVE-003	Info.	Optimized TmpRewardPoolL1::removeFeeRewardPool()	Coding Practices	Fixed
PVE-004	Info.	Inaccurate Event Emitted in StakedPerpToken::stake()	Business Logic	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Missed Access Control in KeeperReward L1/L2

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `KeeperRewardL1.sol`, `KeeperRewardL2.sol`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

For the purpose of incentivizing keepers to help the system to update states (e.g., prices of assets), `KeeperRewardL1/L2` contracts wrap the functions to be invoked by keepers, set rewards for each function, and send keepers `rewardToken` after each successful keeper call. As shown in the code snippet below, the wrapped function is called in line 21 and the rewards are sent to the caller of `updatePriceFeed()` in `postTaskAction()` (line 22).

```
17     function updatePriceFeed(bytes32 _priceFeedKey) external {
18         bytes4 selector = ChainlinkL1.updateLatestRoundData.selector;
19         TaskInfo memory task = getTaskInfo(selector);

21         ChainlinkL1(task.contractAddr).updateLatestRoundData(_priceFeedKey);
22         postTaskAction(selector);
23     }
```

Listing 3.1: KeeperRewardL1.sol

However, as we look into the wrapped function, `ChainlinkL1.updateLatestRoundData()`, we find out that the function is also an external function with no access control. It means if a keeper invokes the wrapped function directly, there's no reward, which makes the state updater have two entries: one with rewards, one without rewards. The same issue is also applicable to `KeeperRewardL2` and `ClearingHouse.payFunding()`.

```
    function updateLatestRoundData(bytes32 _priceFeedKey) external {
        AggregatorV3Interface aggregator = getAggregator(_priceFeedKey);
```

```
        requireNonEmptyAddress(address(aggregator));

        (uint80 roundId, int256 price, , uint256 timestamp, ) = aggregator.
            latestRoundData();
        require(timestamp > prevTimestampMap[_priceFeedKey], "incorrect timestamp");
        require(price >= 0, "negative answer");

        uint8 decimals = aggregator.decimals();

        Decimal.decimal memory decimalPrice = Decimal.decimal(formatDecimals(uint256(
            price), decimals));
        bytes32 messageId =
            rootBridge.updatePriceFeed(priceFeedL2Address, _priceFeedKey, decimalPrice,
                timestamp, roundId);
        emit PriceUpdateMessageIdSent(messageId);
        emit PriceUpdated(roundId, decimalPrice.toUint(), timestamp);

        prevTimestampMap[_priceFeedKey] = timestamp;
    }
```

Listing 3.2: ChainlinkL1.sol

Recommendation Only allow the KeeperRewardL1/L2 contracts to call the underlying functions.

Status As per discussion with the team, they decide to leave it as is to make the state updater functions permission-less.

3.2 Unused Mapping in PerpRewardVesting

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PerpRewardVesting`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `PerpRewardVesting` contract, the `claimWeek()` function allows the caller to claim the vested assets of a specific `_week`. As an enhancement based on Balancer's `MerkleRedeem` contract, a vesting delay (i.e., `vestingPeriodMap[_week]`) is used to prevent the caller from claiming the assets right after those tokens are vested.

```
function claimWeek(
    address _account,
    uint256 _week,
```

```
    uint256 _claimedBalance,
    bytes32[] memory _merkleProof
) public virtual override {
    //
    //          claimableTimestamp       now
    //                  +----------------+
    //                   vesting period
    //  ---------+------+---+------------+--
    //
    //              merkleRootTimestampMap[weeks+1] -> non-claimable//

    // merkleRootTimestampMap[weeks] --> claimable
    //
    uint256 claimableTimestamp = _blockTimestamp().sub(vestingPeriodMap[_week]);
    require(claimableTimestamp >= merkleRootTimestampMap[_week], "Claiming is not yet
        available");
    super.claimWeek(_account, _week, _claimedBalance, _merkleProof);
}
```

Listing 3.3: PerpRewardVesting.sol

As we look into the details of setting the vesting delay, we identify that the `vestingPeriodMap[_week]` is set to a constant value, `defaultVestingPeriod`. Therefore, the `vestingPeriodMap[_week]` mapping is not necessary here. The `claimableTimestamp` in `claimWeek()` could be simply derived by `_blockTimestamp().sub(defaultVestingPeriod)`.

```
    function seedAllocations(
        uint256 _week,
        bytes32 _merkleRoot,
        uint256 _totalAllocation
    ) public virtual override onlyOwner {
        super.seedAllocations(_week, _merkleRoot, _totalAllocation);
        merkleRootTimestampMap[_week] = _blockTimestamp();
        merkleRootIndexes.push(_week);
        vestingPeriodMap[_week] = defaultVestingPeriod;
    }
```

Listing 3.4: PerpRewardVesting.sol

Recommendation Use constant vesting period in `claimWeek()` and remove `vestingPeriodMap`.

Status This issue has been addressed in this commit: 7766c9b.

3.3 Optimized TmpRewardPoolL1::removeFeeRewardPool()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `TmpRewardPoolL1`, `TollPool`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

In `TmpRewardPoolL1` contract, the `removeFeeRewardPool()` function allows the owner to remove a `_token` from the `feeTokens[]` array. While reviewing the implementation, we identify a redundant array traversal which could be optimized. Specifically, as shown in the code snippet below, `isFeeTokenExisted()` is called in line 75 to ensure that the `_token` is existed.

```
73     function removeFeeRewardPool(IERC20 _token) external onlyOwner {
74         require(address(_token) != address(0), "invalid input");
75         require(isFeeTokenExisted(_token), "token does not exist");

77         uint256 lengthOfFeeTokens = getFeeTokenLength();
78         for (uint256 i; i < lengthOfFeeTokens; i++) {
79             if (_token == feeTokens[i]) {
80                 IRewardRecipient feeRewardPool = feeRewardPoolMap[feeTokens[i]];
81                 if (i != lengthOfFeeTokens - 1) {
82                     feeTokens[i] = feeTokens[lengthOfFeeTokens - 1];
83                 }

85                 feeTokens.pop();
86                 delete feeRewardPoolMap[_token];

88                 emit FeeRewardPoolRemoved(address(_token), address(feeRewardPool));
89                 break;
90             }
91         }
92     }
```

Listing 3.5: TmpRewardPoolL1.sol

As shown in the code snippet below, the `feeTokens[]` array is walked through to find the `_token`. However, in line 78, the `removeFeeRewardPool()` walks through the array again and removes the `_token` from the array when `i` reaches the position of the `_token`. Since the second array traversal is necessary, we could simply skip the first one and `revert()` when the second for-loop cannot find the `_token`. This could be done by setting a `found` flag whenever `_token == feeTokens[i]` and `require(found)` in the end of the function.

```
    function isFeeTokenExisted(IERC20 _token) public view returns (bool) {
        for (uint256 i; i < feeTokens.length; i++) {
```

```
            if (_token == feeTokens[i]) return true;
        }
        return false;
    }
```

Listing 3.6: BaseBridge.sol

Same theory applies to `removeFeeToken()` function in the `TollPool` contract.

Recommendation Remove the redundant `isFeeTokenExisted()` call and revise the `removeFeeRewardPool()` function as follows:

```
    function removeFeeRewardPool(IERC20 _token) external onlyOwner {
        require(address(_token) != address(0), "invalid input");
        bool found;
        uint256 lengthOfFeeTokens = getFeeTokenLength();
        for (uint256 i; i < lengthOfFeeTokens; i++) {
            if (_token == feeTokens[i]) {
                found = true;
                IRewardRecipient feeRewardPool = feeRewardPoolMap[feeTokens[i]];
                if (i != lengthOfFeeTokens - 1) {
                    feeTokens[i] = feeTokens[lengthOfFeeTokens - 1];
                }

                feeTokens.pop();
                delete feeRewardPoolMap[_token];

                emit FeeRewardPoolRemoved(address(_token), address(feeRewardPool));
                break;
            }
        }
        require(found, "token does not exist");
    }
```

Listing 3.7: TmpRewardPoolL1.sol

Status This issue has been addressed in this commit: 589bdd5.

3.4 Inaccurate Event Emitted in StakedPerpToken::stake()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StakedPerpToken.sol`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `StakedPerpToken` contract, users are allowed to `stake()` an arbitrary `_amount` of `perpToken`. As a special design, the pending balance of the previous withdrawal (if any) would be re-staked in the `stake()` call. However, while reviewing the implementation, we identify that the `Staked()` event emitted in the end of the `stake()` has an inaccurate amount due to the re-staking.

```
    function stake(Decimal.decimal calldata _amount) external {
        requireNonZeroAmount(_amount);
        address msgSender = _msgSender();

        // copy calldata amount to memory
        Decimal.decimal memory amount = _amount;

        // stake after unstake is allowed, and the states mutated by unstake() will
            being undo
        if (stakerWithdrawPendingBalance[msgSender].toUint() != 0) {
            amount = amount.addD(stakerWithdrawPendingBalance[msgSender]);
            delete stakerWithdrawPendingBalance[msgSender];
            delete stakerCooldown[msgSender];
        }

        // if staking after unstaking, the amount to be transferred does not need to be
            updated
        _transferFrom(perpToken, msgSender, address(this), _amount);
        _mint(msgSender, amount);

        // Have to update balance first
        for (uint256 i; i < stakeModules.length; i++) {
            stakeModules[i].notifyStake(msgSender);
        }

        emit Staked(msgSender, _amount.toUint());
    }
```

Listing 3.8: StakedPerpToken.sol

Specifically, the `Staked()` event is emitted with `_amount` which is the amount of `perpToken` transferred into the `StakedPerpToken` contract. It means the `stakerWithdrawPendingBalance[msgSender]` (if it's greater than 0) amount of `perpToken` are not staked but the corresponding `sPERP` tokens are minted (line 97). We suggest to fix the event emitted to make maintenance easier.

Recommendation Emit `Staked(msgSender, amount.toUint())` in the end of `stake()`.

Status This issue has been addressed by emitting `Staked(msgSender, amount.toUint())` in the end of `stake()` in this commit: 244ff96.

4 Conclusion

In this audit, we have analyzed the Perpetual Protocol's new functionality on Staking, Rewards Vesting, and Keeper. 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.

References

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

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

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

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

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

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

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