SMART CONTRACT AUDIT REPORT

for

PERPETUAL PROTOCOL

Prepared By: Shuxiao Wang

PeckShield

Feb. 23, 2021

Document Properties

Client	Perpetual Protocol
Title	Smart Contract Audit Report
Target	Monotonically changing dynamic K
Version	1.0
Author	Edward Lo
Auditors	Edward Lo, Xudong Shao
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	Feb. 23, 2021	Edward Lo	Final Release
1.0-rc	Feb. 19, 2021	Edward Lo	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 design document and related smart contract source code of the Perpetual Protocol's functionality to monotonically change the dynamic K, 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 audited 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	Feb. 23, 2021

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

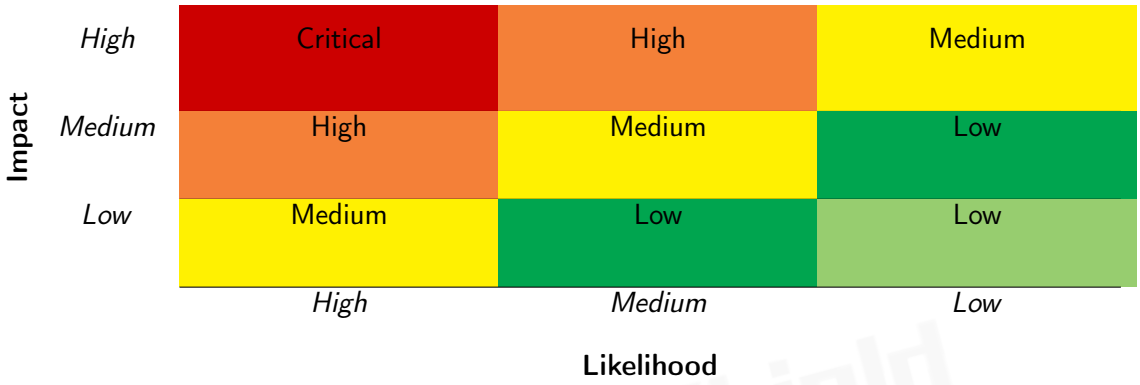
this audit:

- https://github.com/perpetual-protocol/perp-contract (84f9944)

1.2 About PeckShield

PeckShield Inc. [8] 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

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) [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 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 functionality to monotonically change the dynamic K. 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	0	
Informational	4	■■■■
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 4 informational recommendations.

Table 2.1: Key Monotonically changing dynamic K Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Info.	Gas Optimization for Amm::updateOpenInterestBaseAsset()	Business Logic	Fixed
PVE-002	Info.	Misaligned Comparison in Amm::getPendingLiquidityPaymentPerBaseAsset()	Coding Practices	Confirmed
PVE-003	Info.	Erroneous Comment in ClearingHouse::calcLiquidityPayment()	Inaccurate Comments	Fixed
PVE-004	Info.	Inefficient Code Flow in Amm::updateFunding()	Coding Practices	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 Gas Optimization for Amm::updateOpenInterestBaseAsset()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Amm.sol`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

Perpetual protocol allows traders to speculate on the future price of a given asset by buying (going long) or selling (going short) perpetual futures contracts. When traders make changes to their positions (e.g., open or close positions), `ClearingHouse` contract will update open interest status accordingly. As shown in the code snippet below, `updateOpenInterestBaseAsset()` is called when the open interest is changed, and it will call `Amm.updateOpenInterestBaseAsset()` to update `openInterestBaseAsset` in the `Amm` contract.

```
1145    function updateOpenInterestBaseAsset(
1146        IAmm _amm,
1147        Side _openInterestSide,
1148        SignedDecimal.signedDecimal memory _amount
1149    ) internal {
1150        if (_openInterestSide == Side.BUY) {
1151            _amm.updateOpenInterestBaseAsset(_amount, SignedDecimal.zero());
1152        } else {
1153            _amm.updateOpenInterestBaseAsset(SignedDecimal.zero(), _amount);
1154        }
1155    }
```

Listing 3.1: ClearingHouse.sol

```
388    function updateOpenInterestBaseAsset(
389        SignedDecimal.signedDecimal calldata _long,
390        SignedDecimal.signedDecimal calldata _short
391    ) external override onlyCounterParty {
```

```
392         // assume latestOpenInterestBaseAsset is a state with struct type
393         Decimal.decimal memory latestOpenInterestLong = _long.addD(openInterestBaseAsset
                .long).abs();
394         Decimal.decimal memory latestOpenInterestShort = _short.addD(
                openInterestBaseAsset.short).abs();

396         // update values if in the same block
397         uint256 currentBlock = _blockNumber();
398         openInterestBaseAsset.long = latestOpenInterestLong;
399         openInterestBaseAsset.short = latestOpenInterestShort;
400         if (currentBlock != openInterestBaseAsset.blockNumber) {
401             openInterestBaseAsset.blockNumber = currentBlock;
402         }

404         requireMinimalBaseAssetReserve();
405     }
```

Listing 3.2: Amm.sol

As we look into the `ClearingHouse.updateOpenInterestBaseAsset()`, we find out that the function will only update one side at a time (long or short), which means updating long or short is an exclusive operation. However, `Amm.updateOpenInterestBaseAsset()` will update two sides of the `openInterestBaseAsset` no matter what their values are (line 398, 399), which is a waste of gas for setting the same value to the opposite side.

Recommendation Only update long or short side of `openInterestBaseAsset` at a time.

Status This issue has been addressed in this commit: 263cca9.

3.2 Misaligned Comparison in Amm::getPendingLiquidityPaymentPerBaseAsset()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Amm.sol`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

There are two kinds of payment in the Perpetual protocol: premium and liquidity. These payments will be updated by the first transaction of each new block, and traders will realize their payments (if not in migration, only premium payment) on their `swap` / `closePosition` / `removeMargin` operations.

In the `Amm` contract, the `getPendingLiquidityPaymentPerBaseAsset()` function will calculate the liquidity payment per open interest as shown in the code snippet below.

```
1100            if (pendingPremiumPaymentFraction.toInt() == 0) {
1101                Decimal.decimal memory totalOpenInterestBaseAsset =
1102                    openInterestBaseAssetLong.addD(openInterestBaseAssetShort);
1103                paymentOfLong = paymentOfShort = pendingLiquidityPayment.divD(
                        totalOpenInterestBaseAsset);
1104            } else if (isNegativePendingLiquidityPayment == isPositivePremium) {
1105                if (openInterestBaseAssetShort.toUint() > 0) {
1106                    paymentOfShort = pendingLiquidityPayment.divD(
                            openInterestBaseAssetShort);
1107                } else {
1108                    ammPendingLiquidityPayment = pendingLiquidityPayment;
1109                }
1110            } else {
1111                if (openInterestBaseAssetLong.toUint() != 0) {
1112                    paymentOfLong = pendingLiquidityPayment.divD(
                            openInterestBaseAssetLong);
1113                } else {
1114                    ammPendingLiquidityPayment = pendingLiquidityPayment;
1115                }
1116            }
```

Listing 3.3: Amm.sol

The rules can be summarized as the following:

1. Calculate the liquidity payment per base asset (liquidityPayment) has to pay
2. `liquidityPayment` $>$ 0 and `premium` $>$ 0:
   - If there is no long side, `Amm` pays
   - Otherwise, the long side pays
3. `liquidityPayment` $>$ 0 and `premium` $<$ 0:
   - If there is no short side, `Amm` pays
   - Otherwise, the short side pays
4. `liquidityPayment` $<$ 0 and `premium` $>$ 0:
   - If there is no short side, `Amm` receives
   - Otherwise, the short side receives
5. `liquidityPayment` $<$ 0 and `premium` $<$ 0:
   - If there is no long side, `Amm` receives
   - Otherwise, the long side receives

As we look into the details of the code snippet for checking the open interest amount, we identify that the `getPendingLiquidityPaymentPerBaseAsset()` uses different comparison syntax. For checking the short side (line 1105), there is a bigger-than-zero comparison; on the other hand, the long side is checked by a not-equal-to-zero comparison. Though they may have the same result since open interest is a `uint`, using the same comparison semantics is a better practice and will ease future maintenance effort.

Recommendation Use the same comparison semantics.

Status As per discussion with the team, they decide to leave it as is since current comparison has the same result.

3.3 Erroneous Comment in ClearingHouse::calcLiquidityPayment()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ClearingHouse.sol`
- Category: Inaccurate Comments [2]
- CWE subcategory: CWE-1116 [2]

Description

As we introduced in Section 3.2, the liquidity payment is one of the payments that traders will gain or lose. When a trader changes his position, the margin ratio will be re-evaluated with the premium and liquidity payment taken into account. The liquidity payment is calculated in `calcLiquidityPayment()` as shown below.

```
761     function calcLiquidityPayment(IAmm _amm, Position memory _pos)
762         internal
763         view
764         returns (SignedDecimal.signedDecimal memory)
765     {
766         if (_pos.size.toInt() == 0) {
767             return SignedDecimal.zero();
768         }
769         bool isLong = _pos.size.toInt() > 0 ? true : false;
770         SignedDecimal.signedDecimal memory paymentFraction =
771             _amm.getAvgLiquidityPaymentFraction(isLong, _pos.blockNumber);

773         // pos.block will always larger than or equal to current block
774         uint256 blockCount = _blockNumber().sub(_pos.blockNumber);
775         return paymentFraction.mulD(_pos.size.abs()).mulScalar(blockCount);
```

```
776     }
```

Listing 3.4: ClearingHouse.sol

The calculation of the trader's liquidity payment is straightforward by multiplying the average liquidity payment rate , trader's position size, and the block count. Although the math is correct, the comment in line 773 is incorrect: it should be corrected as follows: "the pos.block will always smaller than or equal to current block".

Recommendation Correct the comment

Status This issue has been addressed in this commit: 5276c4c.

3.4 Inefficient Code Flow in Amm::updateFunding()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Amm.sol`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

As we introduced in Section 3.2, there are two kinds of payments and they will be updated when traders take some actions. When these actions are executed by traders, the `ClearingHouse` contract will call `payFunding()` to pay out `Amm`'s funding (premium + liquidity) payment.

```
411     function updateFunding() external override onlyCounterParty returns (SignedDecimal.
            signedDecimal memory) {
412         SignedDecimal.signedDecimal memory ammPendingLiquidityPayment;
413         if (hasPendingMigrationBlocks()) {
414             updateCumuLiquidityPaymentFraction();
415             (, , ammPendingLiquidityPayment) = getPendingLiquidityPaymentPerBaseAsset();
416         }
417         SignedDecimal.signedDecimal memory ammPremiumPayment =
                updatePremiumPaymentFraction();

419         // must after funding payment settled
420         migrateLiquidity();
421         updateLiquidityPaymentUpdatedBlock();
422         return ammPendingLiquidityPayment.addD(ammPremiumPayment);
423     }
```

Listing 3.5: Amm.sol

As shown in the above code snippet, the flow can be summarized as the following:

1. Update liquidity payment (if any)
2. Update premium payment
3. Return Amm's payments (liquidity + premium)

The logic is correct, but we identify a possible improvement as we look into the details of the liquidity payment processing. The Amm contract will only update liquidity payment if necessary (line 413). It will update the global cumulative liquidity payment rate (line 414, details are as follows), then retrieve the Amm's liquidity payment by calling getPendingLiquidityPaymentPerBaseAsset().

```
859     function updateCumuLiquidityPaymentFraction() internal {
860         SignedDecimal.signedDecimal memory liquidityPaymentPerBaseAssetLong;
861         SignedDecimal.signedDecimal memory liquidityPaymentPerBaseAssetShort;
862         (
863             liquidityPaymentPerBaseAssetLong,
864             liquidityPaymentPerBaseAssetShort,

866         ) = getPendingLiquidityPaymentPerBaseAsset();

868         uint256 remainedBlocks = calcRemainedMigrationBlocks();
869         setLatestCumuLiquidityPaymentFraction(
870             liquidityPaymentPerBaseAssetLong.divScalar(remainedBlocks),
871             liquidityPaymentPerBaseAssetShort.divScalar(remainedBlocks)
872         );
873     }
```

Listing 3.6: Amm.sol

In order to update the global cumulative liquidity payment rate, updateCumuLiquidityPaymentFraction() will call getPendingLiquidityPaymentPerBaseAsset() to get the liquidity payment per base asset, which itself is a lengthy computation according to the rules described in Section 3.2. On the other hand, this function also returns the Amm's liquidity payment, in updateFunding() line 415. As we can see here, this function is called twice directly or indirectly in updateFunding(), however, the liquidity payment rate and the Amm's liquidity payment are invariants within this transaction, we don't need to repeat the whole calculation to get the information.

Recommendation Re-factor the above code flow or the function interfaces.

Status This issue has been addressed in this commit: 543b212.

4 | Conclusion

In this audit, we have analyzed the Perpetual Protocol's new functionality to monotonically change the dynamic K, . 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-1116: Inaccurate Comments. https://cwe.mitre.org/data/definitions/1116.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. PeckShield Inc. https://www.peckshield.com.