Security assessment and code review

Initial Delivery: November 2, 2021

Prepared for:
Shao-Kang Lee | Perpetual Protocol
Yenwen Feng | Perpetual Protocol

Prepared by:
Thomas Steeger | HashCloak Inc
Mikerah Quintyne-Collins | HashCloak Inc
Karl Yu | HashCloak Inc

Table Of Contents

Executive Summary

Perpetual Protocol is building a protocol for enabling decentralized and trustless perpetual futures contracts on Ethereum. At the core of their protocol is a concept called the virtual automated market maker (vAMM) that enables one to mint virtual tokens for accounting purposes with no value. In v2 of their protocol, they leverage the increased capital efficiency guarantees of Uniswap V3 in order to build a more capital efficient vAMM.

Perpetual Protocol Team engaged HashCloak Inc for an audit of their Perpetual v2 smart contracts written in Solidity. The audit was done with 3 auditors over a 4 week period, from October 3, 2021 to November 1, 2021.

The version of the codebase that was audited is at commit hash 4edd53fd183ac86f848b38ba22494dddde116456. The scope of the audit were all files that ended in **.sol** with the exception of libraries such as OpenZeppelin's contracts and interfaces for integrating with other protocols such as Chainlink's oracle and Uniswap v3.

During the first two weeks of the audit, we familiarized ourselves with the Perpetual smart contracts and started our manual analysis of the smart contracts. In the final two weeks of the audit, we further investigated the code base, reporting our confusions and concerns to the Perpetual team.

We found a variety of issues ranging from critical to informational.

Severity	Number of Findings
Critical	0
High	1
Medium	3
Low	0
Informational	1

Overview

Perpetual protocol is an on-chain DEX for trading perpetual futures contracts. Perpetual futures contracts are a kind of futures contract in which parties agree to either buy or sell assets at some undetermined point in the future without having their options expire. Further, it allows traders to have long and short positions without having to have custody of the underlying assets.

Perpetual protocol v2 enables traders to trade perpetual futures contracts through its virtual AMM (vAMM) mechanism that uses the newly released Uniswap v3 protocol to guarantee better capital efficiency on trades. The vAMM mints virtual tokens which are virtual versions of supported tokens on Uniswap v3. These virtual tokens hold no value themselves and are solely for accounting purposes.

The Perpetual protocol v2 contracts consists of following main smart contracts:

- Clearing House.sol: Manages of trader's activities such as opening/closing positions and position liquidations. Traders can also add/remove liquidity to/from the vAMM pools.
- Exchange.sol: Manages creating vAMM-based exchanges for Uniswap v3. Traders settle funds with their base tokens, and then swap them for arbitrage in the exchange .
- MarketRegistry.sol: Manages the handling of markets for use within Perpetual Protocol v2 such as setting fees and adding pools.
- Vault.so: Manages and stores all the collateral used throughout the protocol.
- Orderbook.sol: The orderbook is the underlying data structure used in Uniswap v3 for handling buy and sell orders and each orderbook is attached to a respective exchange.
- InsuranceFund.sol : Manages the insurance fund that traders can use to increase their leverage for their positions. Mainly used within the vault contract.

The smart contracts for handling virtual tokens and account balances are

- VirtualToken.sol
- AccountBalance.sol
- BaseToken.sol

Findings

Potential Replay attack on L1 or L2

Type: High
Files affected: MetaTxGateway.sol

On lines 125 to 129, `executeMetaTransaction` requires a signature for both L1 and L2. If Perpetual Protocol were to be deployed on both an L1 and L2 at the same time, there is a potential that transaction could be replayed on both chains.

Impact: Replaying transactions across an L1 and L2 may lead to loss of funds through double spend attacks.
Suggestion: The main way to solve this is to ensure that Perpetual Protocol is deployed only on either L1 or L2 but not both at the same time. If Perpetual Protocol needs to be deployed on both L1 and L2, then ensure that meta transactions contain enough information that ensure that transactions are unique across chains.

Unchecked transfers in uniswapV3MintCallback and uniswapV3SwapCallback

Type: Medium
Files affected: ClearingHouse.sol

On lines 419, 423 and 447, a transfer is made upon either minting or swapping. However, the return value of these transfers is not checked to ensure that these transfers have succeeded or failed.

Impact: An attacker can deposit into the protocol for free potentially leading up to other kinds of attacks that can be executed on the protocol.
Suggestion: Check the return values of each of these transfers and ensure that they revert upon failure.

Unchecked return values in_cancelExcessOrders and removeLiquidity

Type: Medium
Files affected: ClearingHouse.sol

On lines 220 and 522 of ClearingHouse.sol,

`IExchange(_exchange).settleFunding(...)` is called before the rest of the functions are executed. However, since this is an external call, the return value of `settleFunding` should be checked.

Impact: An attacker may be able to cancel excess orders or remove liquidity for free without settling funding payments.
Suggestion: Check the return value of `settleFunding` and revert accordingly.

Equality in deposit should be <=

Type: Medium
Files affected: Vault.sol

When depositing funds into the Vault, special care is taken to take into account deflationary tokens, tokens that charge a fee upon transfers. However, on line 105 of deposit, the following check is made: `balanceBefore.sub(IERC20Metadata(token).balanceOf(from)) == amount`. This makes the assumption that the transfer fee is fixed for this deflationary token and that once the transfer is done that the difference between the before balance and after balance is strictly equal to the amount transferred.

Impact: The balance of the vault for this token will not include the charge fee which may lead to not having enough liquidity being available for this token.
Suggestion: We suggest moving the strict equality to a `<=` in order to take into account the transfer fees.

More elegant baseDebt expressions

Type: Informational
Files affected: AccountBalance.sol

It would be more elegant if we set `baseBalance > 0` as `baseBalance >= 0 .`
As such , when `baseBalance` is equal to 0 , the condition would turn to consider `(-0).toUint256( )` .

Impact: This won't have any real impact on the security of the protocol .
Suggestion: Set `baseBalance >= 0 ? 0 : (- baseBalance ) . toUint256( )` instead of `baseBalance > 0 ? 0 : (- baseBalance ) . toUint256( )` .

General Recommendations

Multiply operations first and then apply division

When dealing with floating/fixed point numbers , we need to do multiplication operations before division in order to reserve as much precision as possible .
Since Perpetual Protocol is backed by SafeMath , we have no worries about overflow problems. As such, we need to minimize loss of precision.

Check address parameters are not 0

It is crucial to check the validity of addresses before using them. For example, we need to check trader addresses that have been used in any functions as input is not zero. This is not done consistently throughout the codebase.

Public vs External Function

Public functions which are solely called from external (other smart contracts or externally owned accounts) should be marked `external` instead of `public`, since it saves gas costs.