Perp V2

Smart Contract Security Assessment

21.12.2021

ABSTRACT

Dedaub was commissioned to perform an audit on the V2 version of the Perp protocol. The Perp protocol is a fully decentralized implementation of perpetual futures, also known as *perps*. Perps have been popularized by the FTX centralized exchange and offer a very high funding rate, which results in highly speculative bets on the prices of underlying cryptocurrencies against a quote token (typically USD). This audit report covers commit hash `a8553acd8ebb42350b9b1c6dd9b73d255b339f57`. Two auditors worked over the codebase over three weeks.

No Critical, High or Medium Severity vulnerabilities were found, which denotes an overall healthy project and codebase. The codebase is very well written and of professional standard. The Perp team provided a lot of helpful documentation material which was vital for the audit, given the high complexity of the project. The audit did not include Uniswap V3 Math libraries and their application, or standard OpenZeppelin code.

Perp V2, which enables decentralized perpetual futures, offers users the ability to gain highly leveraged *price exposure* to synthetic versions of crypto-assets, most notably VETH. The underlying protocol implements a highly complex accounting model, and allows users to trade or stake virtual tokens, via multiple layers of indirection (namely: `ClearingHouse.sol` and `Exchange.sol`), through a standard Uniswap V3 AMM pool initialized with VTokens (synthetic assets).

Perp V2 allows users to hold various positions on one address for multiple virtual base tokens. It should be noted that despite some of these base tokens being correlated with each other, opposite positions in these different tokens would still increase the funding requirements.

Centralization Aspects

As is common in many new protocols, the owner of the smart contracts yields considerable power over the protocol, including changing the contracts holding the user's funds, setting fee ratios, adding addresses to whitelists which could mean adjusting balances of users or minting virtual tokens, etc.

Security Opinion

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e., issues in "regular use") was a secondary consideration, however intensive efforts were made to check the correct application of the mathematical formulae in the reviewed code. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing. Although a number of simulations have been carried out, the crypto-economic effectiveness in a real-world scenario is as yet unknown. Therefore, the financial viability of this protocol in real market conditions cannot be fully established.

A key limitation of this audit, and generally, any exercise intended to establish the security of Perp V2 is that Perp is, admittedly, one of the most complex protocols in DeFi. The accounting logic, and any user action in general involves the reuse of several pieces of complex logic that interact with each other in intricate ways. In addition, Perp V2 has intimate coupling with the internal logic of Uniswap V3 and needs to recalculate and reverse, for instance, (i) fee calculations performed by this AMM in order to account for fees in only the quote token (USD) or (ii) simulate the swapping of assets and calculate slippage.

In terms of architecture, Dedaub notes that there are several design decisions that ensure the economic security of the protocol:

1) The vTokens and vAMM cannot be meaningfully used outside of Perp
2) Clear & separate treatments between realized and unrealized PnL

3) Limits on the amount of slippage that occurs when opening or closing positions.

At the same time, Dedaub notes that there are architectural decisions, and limitations of decentralization, which could potentially threaten the crypto-economic security of Perp V2:

1) Complex Liquidations - given the current implementation, it is hard to predict which positions can be liquidated, since users can be both makers and takers.
2) Complex accounting is hard to verify. Some of the complexity is for making sure that past attacks in Perp V1 cannot be replicated. As a result the team had to also reimplement some parts of Uniswap V3 logic. In addition, some of the checks and limitations introduced may have a negative effect - e.g., limiting slippage only when closing positions means that large positions cannot be efficiently liquidated.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: -User or system funds can be lost when third party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming error.
LOW	Examples: -Breaking important system invariants, but without apparent consequences. -Buggy functionality for trusted users where a workaround exists. -Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Asymmetric advantage for position openers vs. closers/liquidators	**OPEN**
Closing large positions cannot be done in a single block, due to the partial close limitation. However, opening a large position can be done in a single block. This could potentially result in some players opening large positions to gain an advantage. Market participants can potentially exploit this limitation, since they can predict that large changes in the mark price will need to happen in order to close or liquidate large positions.		
L2	`ClearingHouse::liquidate` can be simplified	**OPEN**
In ClearingHouse::liquidate it is required that the trader to be liquidated has no open orders. However, the condition upon which the liquidation is decided, accounts for the		

trader's positions as a maker. This condition could thus be simplified both for clarity and gas efficiency.

```
// CH_CLWTISO: cannot liquidate when there is still order
require(!IAccountBalance(_accountBalance).hasOrder(trader), "CH_CLWTISO");

require(getAccountValue(trader) <
IAccountBalance(_accountBalance).getMarginRequirementForLiquidation(trader
),
      "CH_EAV"
);
```

More specifically, we suggest adding some new functions (essentially variants of getAccountValue and getMarginRequirementForLiquidation as getTakerAccountValue, getTakerMarginRequirementForLiquidation) that will account only for the positions of a trader as a taker. Alternatively, in order to avoid code duplication, another function parameter could be added denoting the accounting only for taker or also for maker positions.

Providing such view functions will make it easier to construct liquidation bots, as the effects of CH::cancelExcessOrders will be decoupled from the liquidation condition. We consider the simplicity of the liquidations-related rules important, since the economic viability of this protocol relies on the ability of liquidators to promptly liquidate user's positions.

L3	Cancel excess orders and liquidate are separate actions	**OPEN**

By having separate calls for closing excess positions and liquidations, liquidator bots can be front-run if they chose to invoke these APIs in separate transactions due to gas limitations. It would be reasonable to cancelExcessOrders and liquidate at once, since only liquidations bring profit to the liquidator.

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	`_isOverPriceLimitBySimulatingClosingPosition` is not needed	**OPEN**

The function above, which re-implements a lot of Uniswap V3's logic down the call stack, can be replaced by the actual Uniswap V3 swap in a separate internal transaction. If the price goes over the limit, the internal transaction could be reverted via exception handlers, and the alternative logic could be exercised instead. This would not only reduce gas requirements but also the complexity of the protocol.

ID	Description	STATUS
A2	Function can be turned into modifier	**OPEN**

The function `ClearingHouseCallee::_requireOnlyClearingHouse` can be turned into a modifier and applied to the calling function for clarity.

ID	Description	STATUS
A3	Debt to InsuranceFund never paid back	**OPEN**

In `Vault::withdraw` some amount might be needed to be borrowed from the InsuranceFund:

```
if (vaultBalanceX10_D < amountX10_D) {
    uint256 borrowedAmountX10_D = amountX10_D - vaultBalanceX10_D;
    IInsuranceFund(_insuranceFund).borrow(borrowedAmountX10_D);
    // Dedaub: debt to InsuranceFund is only growing
    _totalDebt += borrowedAmountX10_D;
}
```

However there is currently no way to pay this debt back.

Also, InsuranceFund contract seems to accrue fees from traders that open positions but there is currently no way to retrieve them:

```
function _openPosition(InternalOpenPositionParams memory params) internal
returns (IExchange.SwapResponse memory) {
      IExchange.SwapResponse memory response =
      IExchange(_exchange).swap(
            IExchange.SwapParams({
            // …
            })
      );

IAccountBalance(_accountBalance).modifyOwedRealizedPnl(_insuranceFund,
response.insuranceFundFee.toInt256())
//…
}
```

A4	Gas savings while `settleFunding`	**OPEN**

In `Exchange::settleFunding`, `fundingGrowthGlobal` and price TWAPs are always calculated as follows:

```
(fundingGrowthGlobal, markTwap, indexTwap) =
_getFundingGrowthGlobalAndTwaps(baseToken);
```

However, for calls within the same block the results will be the same.
Since this function is called at every interaction with the system, it is highly likely to be called more than once in a single block. In order to save gas an if statement could be added:

```
if (timestamp != _lastSettledTimestamp[baseToken]){
    // call _getFundingGrowthGlobalAndTwaps
    // call _updateFundingGrowth
    // update stored vars
```

```
}
else {
    // call _updateFundingGrowth using stored values of fundingGrowth
}
```

A5	Gas usage may increase quadratically to positions	**OPEN**

Whenever a user's position is modified, maintained or liquidated, all of the user's token positions need to be queried. For instance, `AccountBalance::getTotalDebtValue` which gets called on any position action.

Therefore, if we assume that a user with more positions and exposures to more tokens needs to maintain their positions, and the number of actions correlates the number of positions, the gas usage really scales quadratically for such a user.

A6	No functionality to remove pools	**INFO**

We recommend the addition of a function `MarketRegistry::removePool`, to complement the others.

A7	Current design doesn't allow for multiple pools of the same base token	**INFO**

There are several cases where the base token is considered as a unique id for the underlying pool. For example:

```
address pool = IMarketRegistry(_marketRegistry).getPool(params.baseToken);
uint256 feeGrowthGlobalX128 = _feeGrowthGlobalX128Map[params.baseToken];
int24 tick = _getTick(baseToken);
```

This is sufficient as long as the protocol only supports a single quote token for all pools but will be a problem in case the protocol is to be extended.

A8	Compiler known issues	INFO
The contracts were compiled with the Solidity compiler v0.7.6 which, at the time of writing, have some known bugs. We inspected the bugs listed for this version and concluded that the subject code is unaffected.		

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.