Perpetual Protocol

Smart Contract Security Assessment

13.09.2022

ABSTRACT

Dedaub was commissioned to perform a security audit of multiple new releases of the Perpetual V2 protocol and its veRERP implementation.

The scope of the audit focused mainly on the changes introduced in the new versions of the Perpetual V2 protocol available at the at the time private repository https://github.com/perpetual-protocol/perp-lushan, and the implementation of vePERP available at https://github.com/perpetual-protocol/voting-escrow.

The specific versions of the Perpetual V2 protocol whose changes were considered in the audit are:

- v2.2.0-rc (601bd63c918401e55222aebf2b1dc13d9a9ecbae)
  - ClearingHouse.sol
  - InsuranceFund.sol
  - Vault.sol
- v2.2.1-rc (8182424d353456c8bac5ddee53c5df8b9055ec16)
  - ClearingHouse.sol
  - InsuranceFund.sol
- v2.3.0-rc (ac8e436d41de7f030c5244eea4a65106fc74297c)
  - InsuranceFund.sol
- v2.4.0-rc (d7e896a44a55cd140a82aa6bcccd29a712c5ed4c)
  - CollateralManager.sol
  - Vault.sol

The implementation of vePERP considered the following contracts up to commit 0b1ae23c3da120b0abf40bc7425ac1da55f744ae:

- vePERP.vy
- FeeDistributor.vy
- SurplusBeneficiary.sol

Two auditors worked over the codebase over 2 weeks.

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.

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:

ID	Description	STATUS
M1	Bad debt soft circuit break can be circumvented	**OPEN**

Version 2.2.1-rc of the protocol introduces a soft circuit break, which should disallow all trades/liquidations that generate bad debt when the insurance fund has been already depleted.

Bad debt is settled by calling the public function `Vault::settleBadDebt`, which is called by `ClearingHouse::liquidate` and `Vault::liquidateCollateral`. `ClearingHouse::liquidate` implements a requirement check that ensures the insurance fund has enough settlement token balance to cover the bad debt incurred to the protocol by the liquidation. Otherwise the liquidating transaction will revert. However, `Vault::liquidateCollateral` does not implement such a requirement, which may allow liquidations of collateral that under certain conditions generate bad debt.

We have identified two such scenarios:

1. A trader closes her unfavorable position leading to negative PnL but cannot yet get liquidated due to the value of their non-settlement collateral. The value of

the trader's collateral decreases and can no longer cover the negative PnL allowing anyone to liquidate her. Another trader calls `Vault::liquidateCollateral` to liquidate the trader, incurring bad debt to the system in case the insurance fund is not able to cover the debt at the moment, as there is no check against that.

2. A trader that is using non-settlement tokens as collateral has her position gotten liquidated. Due to the fact that the function `Vault::settleBadDebt` requires the trader to not have non-settlement collateral the bad debt will not be settled before liquidating the trader for every non-settlement collateral they have deposited in her Vault. Usually a liquidator will liquidate the trader's position and all their non-settlement collateral in a single transaction (using a specially developed smart contract). However, if this does not happen, the requirement in `ClearingHouse::liquidate`, which ensures that the insurance fund can cover the debt, is deemed useless as the debt might increase between the transaction that liquidates the position and the transaction(s) that liquidate the collateral(s), letting the insurance fund unable to cover it in whole.

LOW SEVERITY:

ID	Description	STATUS
L1	SurplusBeneficiary::setFeeDistributor does not remove the infinite approval for _token given to the old fee distributor.	**OPEN**

`SurplusBeneficiary::setFeeDistributor` sets the new fee distributor contract and approves it to be able to transfer an infinite amount of USDC. However, the approval of the old fee distributor is not revoked, allowing it to transfer any amount of USDC even though that contract might have been deemed obsolete or even vulnerable.

OTHER/ ADVISORY ISSUES:

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

A1	ERC20::transfer might get called with amount set to 0	**OPEN**

SurplusBeneficiary::dispatch computes the amount of USDC that should be transferred to the treasury and executes the transfer without checking first that the transferred amount is not 0.

```
function dispatch() external override nonReentrant {
  // ..

  uint256 tokenAmountToTreasury =
    FullMath.mulDiv(tokenAmount, _treasuryPercentage, 1e6);

  // Dedaub: tokenAmountToTreasury might be 0 due to _treasuryPercentage
  // being 0 or due to rounding.
  SafeERC20.safeTransfer(IERC20(token), _treasury, tokenAmountToTreasury);

  // ..
}
```

`oldBalance` and `newBalance` are equal when

A2	SurplusBeneficiary::_token can be declared immutable	**INFO**

Storage variable `_token` of the SurplusBeneficiary contract could be declared immutable, which would reduce the gas required to access it, as it is only set in the contract's constructor.

A3	set* functions should ensure new value is not equal to old	**INFO**

Functions `setFeeDistributor` and `setTreasury` of the SurplusBeneficiary contract could implement a check that ensures the new value, which is being set, is not equal to the old one.		
A4	Infinite USDC approval given to the FeeDistributor contract	**INFO**
When setting the FeeDistributor contract for the SurplusBeneficiary, infinite USDC approval is also given to it. An alternative approach would be to set the approval (in function `SurplusBeneficiary::dispatch`) to the amount transferred prior to every transfer happening to avoid the dangers that come with approving a contract for an infinite amount. Of course, there is a tradeoff; the extra approve call happening in every call of `dispatch` would translate in higher gas costs, which could be considered bearable as the protocol is deployed on Optimism.		
A5	Whitelist debt threshold can be set to lower than the default	**INFO**
There is no check to ensure the whitelist debt threshold cannot be set to a value that would be less than the default debt threshold. This might be intentional but the term “whitelist” could have users expect that their debt threshold can only increase from the default.		
A6	Compiler known issues	**INFO**
The contracts were compiled with the Solidity compiler v0.7.6 which, at the time of writing, has a few known bugs. We inspected the bugs listed for this version and concluded that the subject code is unaffected.		

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, killing contracts (FeeDistributor), using emergency unlock (vePERP)etc.

In addition, the owner of the protocol has total control of several protocol parameters:

- the treasury contract address
- the percentage of funds going to the treasury
- the fee distributor contract address
- the insurance fund surplus threshold
- the insurance fund surplus beneficiary contract
- the whitelisted debt threshold

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, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure the most prominent protocols in the space. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Prominent blockchain protocols hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.