Perpetual Protocol

Smart Contract Security Assessment

27.04.2022

ABSTRACT

Dedaub was commissioned to perform a security audit of the new multi-collateral Vault of the Perpetual V2 protocol.

The scope of the audit focused mainly on the Vault and CollateralManager contracts of the at the time private repository https://github.com/perpetual-protocol/perp-lushan, up to commit `6da1e589d4be3e28588b0a96460b0ed777261123`. Two auditors worked over the codebase over 8 days.

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.

The scope of this audit was limited to the collateral vault and its liquidation mechanism. There are desirable characteristics to the way multi-collateral support has been designed in Perpetual Protocol. Mainly, (1) it reduces the reliance of position liquidations, (2) improves overall UX, and finally, (3) it reduces the risk of bad debt. Regarding (1) it has been recognized several times, both in our audits and also by the team that position liquidations are especially problematic, due to complexity when triggering, gas requirements and in some cases bypassing of price checks (e.g., the liquidation slippage sandwich attack).

On the other hand, after this change, users and other services interacting with Perpetual Protocol should be aware that there are 3 separate liquidation mechanisms (for Makers, for Takers and for Collateral). In turn, each mechanism might require several steps,

perhaps in separate blocks, with implicit ordering between these steps. For instance, a user that has both a maker and taker position, and multiple forms of collateral may need to be liquidated multiple times. In this case, first by liquidating each collateral type, then canceling the orders, and finally liquidation of taker positions. Liquidation of taker positions may also require multiple steps if the position is large enough. Monitoring the behavior of these liquidations under market conditions may reveal that there are limited liquidation market participants, due to the complexity of liquidations.

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" or "acknowledged" but no action taken, 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	`Vault::depositFor` can be used to prevent a user from depositing	**DISMISSED**

Vault allows users to make deposits for another user via the `depositFor` method. This functionality in combination with the limit imposed on the number of different collateral tokens a user can deposit could be exploited by an adversary to prevent the user from depositing into their Vault.

The parameter imposing the limit on the different collaterals a user can deposit is defined in `CollateralManager` as `_maxCollateralTokensPerAccount` and is used in `Vault::_modifyBalance`, which is called by `Vault::deposit`.

The attack could be a griefing attack, i.e., just making the user's life harder by forcing them to withdraw the irrelevant collateral before being able to deposit, but it could also be much more serious. For example, an attacker could target a user that is close to becoming liquidatable and is trying to deposit a number of new collateral tokens to prevent their liquidation. The attacker could frontrun the user and deposit tiny amounts of different collateral tokens on behalf of them using the method depositFor until no

further deposits are possible. By doing this the attacker would greatly increase the chance of the user becoming liquidatable and they could then proceed with the liquidation to make a nice profit.

LOW SEVERITY:

ID	Description	STATUS
L1	`Vault::receive` allows any `msg.sender` to send Ether	**RESOLVED**

ETH deposited into the Vault contract is converted to WETH by being deposited into the WETH contract. A user wishing to withdraw their ETH needs to call the `withdrawEther` method, which in turn calls the `withdraw` method of the WETH contract. As part of the unwrapping procedure of WETH, ETH is sent back to the Vault contract, which needs to be able to receive it and thus defines the special `receive()` method. It is expected (mentioned in a comment) that the `receive()` method will only be used to receive funds sent by the WETH contract. However, there is no check enforcing this assumption, allowing practically anyone to send ETH to the contract. We believe that the current version of the code is not susceptible to any attacks that could try to manipulate the accounting of ETH performed by the Vault. Still, we cannot guarantee that no attack vectors will arise as the codebase evolves and thus suggest adding a check on the msg.sender as follows:

```
receive() external payable {
    require(_msgSender() == _WETH9, "msg.sender is not WETH");
}
```

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	Vault allows 0 value withdrawals	**ACKNOWLEDGED**

The Vault contract allows 0 value withdrawals through its external `withdraw` and `withdrawEther` methods. We believe that adding a requirement that a withdrawal's amount should be greater than 0 would improve user experience and prevent the unnecessary spending of gas on user error.

[The suggestion has been acknowledged by the protocol's team and might be implemented in a future release.]

A2	Vault allows 0 value liquidations	**ACKNOWLEDGED**

The Vault contract allows 0 value liquidations through its `liquidateCollateral` method. Disallowing such liquidations will protect users from unnecessarily spending gas in case they make a mistake.

[The suggestion has been acknowledged by the protocol's team and might be implemented in a future release.]

A3	`Vault::_modifyBalance` gas optimization	**RESOLVED**

Internal method `Vault::_modifyBalance` allows the `mount` parameter to be 0. This behavior is intended, as it is clearly documented in a comment. Nevertheless, when `amount` is 0, no changes are applied to the contract's state, as can be seen below:

```
function _modifyBalance(
    address trader,
    address token,
    int256 amount
) internal {
```

```
    // Dedaub: code has no effects on storage, still consumes some gas
    int256 oldBalance = _balance[trader][token];
    int256 newBalance = oldBalance.add(amount);
    _balance[trader][token] = newBalance;

    if (token == _settlementToken) {
        return;
    }

    // register/deregister non-settlement collateral tokens
    if (oldBalance != 0 && newBalance == 0) {
        // Dedaub: execution will not reach here when amount is 0
        // ..
    } else if (oldBalance == 0 && newBalance != 0) {
        // Dedaub: execution will not reach here when amount is 0
        // ..
    }
}
```

`oldBalance` and `newBalance` are equal when `amount` is 0, thus no state changes get applied. Still some gas is consumed, which can be avoided if the method is changed to return early if `amount` is 0.

A4	`Vault::_getAccountValueAndTotalCollateralValue` gas optimization	**RESOLVED**

Method `_getAccountValueAndTotalCollateralValue` calls the AccountBalance contract's method `getPnlAndPendingFee` twice, once directly and once in the call to `_getSettlementTokenBalanceAndUnrealizedPnl` in `_getTotalCollateralValue`. The first call to `getPnlAndPendingFee` to get the unrealized PnL could be removed if the code was restructured appropriately to reuse the same value returned by `_getSettlementTokenBalanceAndUnrealizedPnl`.

A5	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 and adding tokens, which potentially means borrowing tokens using fake collateral, etc.

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

- the collateral ratio of tokens
- the discount ratio (applicable in liquidation)
- the deposit cap of tokens
- the maximum number of different collateral tokens for an account
- the maintenance margin buffer ratio
- the allowed ratio of debt in non settlement tokens
- the liquidation ratio
- the insurance fund fee ratio
- the debt threshold
- the collateral value lower (dust) limit

In case the aforementioned parameters are decided by governance in future versions of the protocol, collateral ratios should be approached in a really careful and methodical way. We believe that a more decentralized approach would be to alter these weights in a specific way defined by predetermined formulas (taking into consideration the on-chain volatility and liquidity available on-chain) and allow only small adjustments by governance.

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.