Preon Protocol

Security Assessment

June 30, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of the Preon protocol, borrowing elements mainly from Gravita (an adaptation itself of the Liquity code), but also integrating functionality from other DeFi code bases. The audit found the code to be very competently integrated or freshly developed, but cautions on some aspects, both protocol-level and local in code.

SETTING & CAVEATS

The audit was performed at commit hash `42802f2c09e24f560a187dacc9ff81e2cae7fd16` of the preon-contracts repository.

An audit team of two auditors worked on the codebase for one working week.

Per our agreement with the client, the audit was conducted as a delta audit over the Gravita codebase, at commit `c23aed49408681dc760c5c61ff102f1913ce5adb`.

In more detail, the following contracts were minorly edited relative to the Gravita codebase, and only the edits were thoroughly inspected:

- ActivePool: adds vaults for ERC20 tokens, functions such as depositERC20;
- AdminContract: just adds `treasury` public field, which is used for gas compensation in VesselManagerOperations;
- BorrowerOperations: calls the depositERC20 in active pool, otherwise just formatting;
- CollSurplusPool: only formatting;
- DebtToken: adds OFT functionality, otherwise unchanged;
- DefaultPool: only formatting;
- FeeCollector: only minor change is approve-transferFrom instead of direct transfer pattern for FeeDistributor;
- PriceFeed: only formatting;

- SortedVessels: only formatting;
- StabilityPool: adds setter for communityIssuance, otherwise only formatting;
- Timelock: only non-functional changes;
- VesselManager: only formatting;
- VesselManagerOperations: liquidator of Vessels does not get gas compensation;
- GRVT/CommunityIssuance: only formatting;
- GRVT/GRVTToken: only formatting;
- GRVT/LockedGRVT: only formatting;
- Pricing/FixedPriceAggregator: only formatting;
- Pricing/WstEth2EthPriceAggregator: only formatting;
- Dependencies:
    - GravitaMath/PreonMath: only formatting;
    - GravitaBase/PreonBase: only formatting.

The underlying code of the above files was treated as trusted, although we did happen to notice two issues in contracts under the GRVT folder, which seems unused in the Gravita protocol. Thus, this part of the Gravita codebase should be treated as less reliable than others.

The following contracts were exhaustively audited:

- OptionsToken
- BalancerOracle
- PreonOFT.

Much of this code derives from past codebases (Radiant, Timeless Finance), however we did not have enough prior experience with them.

Finally, functionality in the VoteEscrow directory was explicitly outside the audit scope. This functionality was derived from the Dystopia exchange code and is considered trusted. However, we caution that we have not considered this functionality even in terms of whether it integrates well with the rest of the codebase.

The above is also a general caveat for delta audits and audits over codebases that integrate separately-developed protocols. The developers should exercise extreme caution and test very thoroughly, both at the unit level and at the integration level, to ensure functional correctness. (We note that the project already includes good unit tests and overall seems competently developed.) This consideration is magnified for multi-chain deployment.

As part of the audit, we also reviewed the fixes for the issues included in the report. The fixes were delivered at commit `8dcddc6946d3fa99bd11b9487eda74007a0ab187`. The code at this commit was inspected only locally, over the changes that pertain to this report's items. Per the earlier caveat (about local inspection of changes), locally sound fixes may have issues in a broader context. The development team is advised to be especially vigilant with testing the consequences of fixes performed after the initial audit.

We also point to our audit of the majority of the Gravita functionality for deployment considerations, e.g., on the tokens that can be safely used. We repeat the most relevant of these considerations in this document, with the prefix [Gravita consideration], but skip several others, especially at "Advisory" severity, for brevity.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Although a high-level specification describing interactions within the protocol was provided, functional correctness (i.e. issues in "regular use") was a secondary consideration. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

PROTOCOL-LEVEL CONSIDERATIONS

The following considerations are repeated verbatim from our audit of the Gravita codebase. We are listing them just to raise awareness. The Preon protocol team has already discussed with us the rationale over keeping the first of these decisions.

- There is a REDEMPTION_SOFTENING_PARAM set to 97%. This has the effect that third-party (not the vessel's owner) redeemers only receive 97% of the collateral. As a result, the STAR (*was*: GRVT) token will likely not be well-pegged to the USD, but trade slightly lower.
- Unlike in Liquity, the `baseRate` quantity is used to adjust the fees only for redemptions and NOT for borrowing. (In Liquity, if there are recent redemptions, there are higher fees both for redemptions and for borrowing: https://github.com/liquity/dev#intuition-behind-fees)

  In Gravita, the `baseRate` is per-asset, and, thus, only affects redemptions. The cost of borrowing is entirely unaffected by whether there have been recent redemptions: the formula for borrowing fees just includes the static per-asset fee, not the dynamically-changing `baseRate`.

  This is a fine design choice, but it is clearly a matter of financial protocol design.
- The Gravita fees calculation permits limited economic gaming. For instance, fee-wise it may be preferable for a client to open a new debt position instead of increasing his/her old one, just because of the fee refund curve, which remains linear.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting 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 a programming error.
LOW	Examples: • Breaking 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. "Info" items are *a priori* not necessary to address, as long as the protocol is aware of the potential issue.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	The caller of `OptionsToken::exerciseAndZap` is not protected against slippage	**RESOLVED**

The call to `OptionsToken::exerciseAndZap` can be front-run (with the risk varying per network) to "tilt" the liquidity pool before joining, so that the funds that enter the pool appear worthless, at great loss to the caller of the function. Subsequent untilting of the pool can yield gains for the attacker. A function parameter that allows specifying the minimum expected amount of the liquidity pool (`preonBpt`) token should be added, to provide protection to callers.

```
function exerciseAndZap(
      uint256 amount,
      address recipient
) external virtual returns (uint256 bptAmount) {
...
      uint256 _before = preonBpt.balanceOf(recipient);
...
      balancerVault.joinPool(
            preonBpt.getPoolId(),
            address(this),
            recipient,
            IVault.JoinPoolRequest({
            assets: assets,
            maxAmountsIn: maxAmountsIn,
            userData: abi.encode(1, maxAmountsIn, 0),
            fromInternalBalance: false
            })
      );

      bptAmount = preonBpt.balanceOf(recipient) - _before;
```

```
        // Dedaub: this amount is never checked against anything

        emit ExerciseAndZap(msg.sender, recipient, amount, bptAmount);
}
```

MEDIUM SEVERITY:

ID	Description	STATUS
M1	[Gravita consideration] Reentrancy considerations	**INFO**

We recommend the use of the protocol only with tokens that do not have callbacks to untrusted parties, to completely mitigate all reentrancy concerns. However, the Gravita codebase should already be hardened against reentrancy (for "usual" callbacks), so the main concern is with "rare" callback patterns.

Namely, the protocol should not be used with collateral tokens that perform callbacks to the **sender** of tokens. E.g., ERC 777 tokens have this behavior and they are well-known as reentrancy death traps. Since these tokens are very rare, this is not a true limitation.

The specific threat with such tokens is that the internal function BorrowerOperations::_activePoolAddColl (which does a `transferFrom` from an untrusted party) is used in the middle of code that has complex effects after the function call. Viewed differently, such tokens are very different from ETH in callback behavior, and since the Liquity codebase of origin has been written and tested with ETH in mind, such unintuitive callbacks can yield major violations of the checks-effects-interactions pattern.

```
function _activePoolAddColl(
```

```
        address _asset,
        IActivePool _activePool,
        uint256 _amount
) internal {
        IERC20Upgradeable(_asset).safeTransferFrom(
                msg.sender,
                address(_activePool),
                SafetyTransfer.decimalsCorrection(_asset, _amount)
        );
        _activePool.depositERC20(_asset, _amount);
}

// called in:

function openVessel(
        address _asset,
        uint256 _assetAmount,
        uint256 _debtTokenAmount,
        address _upperHint,
        address _lowerHint
) external override {
   …
   _activePoolAddColl(vars.asset, contractsCache.activePool, _assetAmount);
   _withdrawDebtTokens(
                vars.asset,
                contractsCache.activePool,
                contractsCache.debtToken,
                msg.sender,
                _debtTokenAmount,
                vars.netDebt
  );
  // Move the debtToken gas compensation to the Gas Pool
  _withdrawDebtTokens(
                vars.asset,
                contractsCache.activePool,
                contractsCache.debtToken,
                gasPoolAddress,
                adminContract.getDebtTokenGasCompensation(vars.asset),
                adminContract.getDebtTokenGasCompensation(vars.asset)
  );
```

```
  …
}
```

M2	The protocol does not force the users to submit the correct deadline when they exercise an option	**DISMISSED**

Function `OptionsToken::exerciseWithDeadline` seems to serve no purpose, relative to function `exercise`. The deadline is supplied by the external caller themselves, thus conferring no guarantees at all.

```
function exercise(
      uint256 amount,
      uint256 maxPaymentAmount,
      address recipient
) external virtual returns (uint256 paymentAmount) {
      return _exercise(amount, maxPaymentAmount, recipient);
}
function exerciseWithDeadline(
      uint256 amount,
      uint256 maxPaymentAmount,
      address recipient,
      uint256 deadline
) external virtual returns (uint256 paymentAmount) {
      if (block.timestamp > deadline) revert OptionsToken__PastDeadline();
      return _exercise(amount, maxPaymentAmount, recipient);
}
```

M3	Accounting error in `LockedPREON::lowerEntityVesting`	**RESOLVED**

This function is intended to decrease the `entitiesVesting[_entity].totalSupply` to `newTotalSupply`. (It would also be nice if a check that the `newTotalSupply` is indeed lower than the old one were added.) This change in the total supply should

additionally decrease the `assignedPREONToken` variable by `entitiesVesting[_entity].totalSupply - newTotalSupply`.

LOW SEVERITY:

ID	Description	STATUS
L1	`preonDistribution` is treated as constant in `CommunityIssuance::_getLastUpdateTokenDistribution()`	**DISMISSED (protocol will be aware of the potential issue)**
The function `_getLastUpdateTokenDistribution`, as its name suggests, computes the total distributed tokens since tha last update, using the formula: `uint256 totalDistribuedSinceBeginning = preonDistribution * timePassed;` where `preonDistribution` is the weekly distribution of tokens. The problem is that `preonDistribution` is not necessarily constant for the whole `timePassed` from `lastUpdateTime` to `block.timestamp`, because the controller of the contract can change its value calling `setWeeklyPreonDistribution` and this should be reflected in the formula.		
L2	The use of OFTV2 does not seem well-motivated	**INFO**
Although this will likely not cause issues, the use of base contract OFTV2 (to give multi-chain capabilities) is going against the LayerZero integration guidelines, which state: *For bridging only between EVM chains use OFT and for bridging between EVM and non EVM chains (e.g., Aptos) use OFTV2.*		

The Preon project is explicitly targeting EVM chains, per our communication with the development team.

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	[Gravita consideration] Whitelisted contracts can mint arbitrarily large amounts of debt tokens	**INFO**

The role of the whitelisted contracts is not completely clear to us. There is only one related comment in `DebtToken.sol` :

```
// stores SC addresses that are allowed to mint/burn the token (AMO strategies,
// L2 suppliers)
mapping(address => bool) public whitelistedContracts;
```

These contracts can mint debt tokens without depositing any collateral calling DebtToken::mintFromWhitelistedContract. This could be a serious problem if such a contract was malicious. Also, even if these contracts work as expected, minting debt tokens without providing any collateral could have a serious impact on the price of the debt token.

N2	Protocol owners can set crucial parameters	**INFO**

Key functionality is trusted to the owner of various contracts. Owners can set the kinds of collateral accepted, the oracles that are used to price collateral, etc. Thus, protocol owners should be trusted by users.

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	[Gravita consideration] Tokens with more than 18 decimals are not supported	**INFO**

Tokens with more than 18 decimals are not supported, based on the SafetyTransfer library (outside the audit scope).

```
function decimalsCorrection(address _token, uint256 _amount)
       internal
       view
       returns (uint256)
    {
       if (_token == address(0)) return _amount;
       if (_amount == 0) return 0;

       uint8 decimals = ERC20Decimals(_token).decimals();
       if (decimals < 18) {
             return _amount.div(10**(18 - decimals));
       }

       return _amount; // Dedaub: more than 18 not supported correctly!
    }
```

We do not recommend trying to address this, as it may introduce other complexities for very little practical benefit. Instead, we recommend just being aware of the limitation.

A2	`updateAssetVault` does not allow real updating	**INFO**

In ActivePool, the vault for a certain asset cannot be changed once set. This is a limitation and inconsistent with the function name, which is “update” and not “set”.

```
function updateAssetVault(
      address _asset,
      address _vault
) external onlyOwner {
      require(_asset != address(0), "ActivePool: Invalid asset address");
      require(_vault != address(0), "ActivePool: Invalid vault address");
      require(
            assetVault[_asset] == address(0),
            "ActivePool: Vault already set"
      );
      assetVault[_asset] = _vault;
      emit AssetVaultUpdated(_asset, _vault);
}
```

A3	Magic constants	**INFO**

Our recommendation is for all numeric constants to be given a symbolic name at the top of the contract, instead of being interspersed in the code.

- PreonOFT::setFee:

```
require(_fee <= 1e4, "Invalid ratio");
```

A4	Compiler bugs	**INFO**

The code has the compile pragma ^0.8.17. For deployment, we recommend no floating

pragmas, i.e., a fixed version, for predictability. (Note specifically that 0.8.20 is a known multi-chain threat, due to use of the PUSH0 opcode, which most chains do not recognize. The ^0.8.17 pragma would allow 0.8.20.)

Solc version 0.8.17, specifically, has no known bugs, at the time of writing.

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 some of the most prominent protocols in DeFi. 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. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.