# Vesper Synth delta audit

Smart Contract Security Assessment

Feb 16, 2022

## ABSTRACT

Dedaub was commissioned to perform a security audit of several smart contract modules of the Vesper Synth protocol.

The scope of the audit included the most recent updates and fixes to the contracts in the repository https://github.com/bloqpriv/vesper-synth, from the commit 143165906c47ce152826cfa3dcd028d0784b5b8d of the previous audit, up to commit
. A brief description of the included PRs between these commits is below:

- PR #241: Allow swaps even when the account's position is unhealthy
- PR #245,#255: Use a MasterOracle contract to allow different oracles for each asset, and other oracle improvements.
- PR #252: Store the non-zero deposit and debt tokens of each account, to avoid looping over the full array of assets.
- PR #259: Allowance checks in DepositToken.transferFrom()
- PR #261: The Collateralization Ratio now depends on the deposit token, not on the debt.
- PR #280: Add nonReentrant guard in DepositToken::_transfer.
- PR #281: Fix DebtTokenUpgrader.

This was a delta audit of the above changes. Nevertheless, since the protocol's codebase remains relatively small, one auditor (who had not previously seen the code) audited the whole protocol from scratch, and another auditor (who had participated in the previous audit) examined more deeply the current delta.
The auditors considered protocol-level attacks, the general pricing model, and other major considerations in the overall architecture. We reviewed the code in significant

depth, assessed the economics of the protocol and processed it through automated tools. We also decompiled the code and analyzed it, using our static analysis (incl. symbolic execution) tools, to detect possible issues.

## Setting and Caveats

The core of the Vesper Synth protocol is robust, with a clean architecture and reasonably simple, well-organized code. The Controller contains most of the core logic, with the accounting cleanly split in the DepositToken, DebtToken and SyntheticAsset contracts, allowing for modularity but with limited code complexity. Due to the limited audit time relative to the size of the code base, the auditors strategically focused more on the core contracts (all contracts at the root of the `/contracts` directory) although they did also review auxiliary contracts (`/contracts/oracle`, `/contracts/upgraders`, etc.).

It should be noted that a major attack target of  Vesper Synth are the underlying price oracles (as with any protocol that heavily relies on oracles), which pose unpredictable threats that are hard to assess in the audit. We suggest paying special attention to all aspects of these oracles, and include some specific recommendations in this audit.

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") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. 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:<br>-User or system funds can be lost when third party systems misbehave.<br>-DoS, under specific conditions.<br>-Part of the functionality becomes unusable due to programming error. |
| LOW | Examples:<br>-Breaking important system invariants, but without apparent consequences.<br>-Buggy functionality for trusted users where a workaround exists.<br>-Security issues which may manifest when the system evolves. |

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

## CRITICAL SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| C1 | DoS caused by convertToUsd/convertFromUsd failing for tiny amounts | **RESOLVED** |

The convertToUsd/convertFromUsd methods in MasterOracle/DefaultOracle revert if the converted amount is 0.

```
require(_amount > 0, "invalid-price");
```

However, converting a tiny source amount (e.g., 1 wei) might lead to the target amount being 0, depending on the underlying oracle (we tested that this indeed happens for ChainlinkPriceProvider). As a consequence, having either a deposit or debt of a tiny amount, in just a *single asset*, will cause Controller::debtOf / depositOf / debtPositionOf to fail (even if the account holds multiple other assets).

```
for (uint256 i = 0; i < depositTokensOfAccount.length(_account); ++i) {
  IDepositToken _depositToken = IDepositToken(depositTokensOfAccount.at(_account, i));
  uint256 _amountInUsd = oracle.convertToUsd(_depositToken,
    _depositToken.balanceOf(_account)); // THIS MIGHT CAUSE THE WHOLE LOOP TO REVERT
  _depositInUsd += _amountInUsd;
  _mintableLimitInUsd += _amountInUsd.wadMul(_depositToken.collateralizationRatio());
}
```

Given that debtPositionOf is used in almost all functionalities of the protocol, this can lead to a variety of DoS attack scenarios, including the ones below (and possibly others).

1. **Preventing liquidation.**
   Assume that an adversary has an unhealthy loan that he cannot repay, and he wants to avoid the liquidation of his collateral deposit (in token D). He can deposit 1 wei of some other token A, causing Controller::liquidate (which calls method debtPositionOf) to fail. Whenever he wishes, in the future, he can deposit a larger amount of token A, repay his loan and withdraw his deposit. (Note that a third-party can still liquidate the adversary's deposit by performing

a `deposit` of token A themselves, `_onBehalfOf` the adversary. But this requires a deep understanding of why liquidation failed in the first place, possession of token A, and it can be made even harder by depositing tiny amounts of multiple tokens.)

2. **Withdrawal DoS via deposit.**
   An adversary can deposit 1 wei of some token A, `_onBehalfOf` a victim. This will cause `Controller::withdraw` to fail, preventing the victim from withdrawing his deposit (in some other token D). The victim can recover by depositing token A himself, but this requires an understanding of the problem, and possession of token A (recovery can be made even harder by depositing tiny amounts of multiple tokens).
3. **Withdrawal DoS via repay.**
   A DoS can be similarly achieved by having a really small *debt*, instead of a deposit. This is protected by `debtFloorInUsd`, which however is optional. If `debtFloorInUsd` is not set, an adversary can `repay` the debt of a victim (which could be small), leaving only a tiny amount of debt. This will cause the `Controller::withdraw` method to always fail, preventing the victim from withdrawing their deposit.
   Moreover, although a debt-based DoS is harder to achieve (due to the `debtFloorInUsd` parameter and the need to repay the loans), it is also *impossible to recover from*! This is due to the fact that the user needs to increase their debt, and not their deposit. However, `Controller::mint` no longer works, since it calls `debtPositionOf`! Note that repaying the remaining debt is also not possible.

We recommend modifying `MasterOracle/DefaultOracle` to treat the case of a zero target amount gracefully. Moreover, making `debtFloorInUsd` mandatory, and/or possibly introducing a similar `depositFloorInUsd` could be considered.

## HIGH SEVERITY:

[No high severity issues]

## MEDIUM SEVERITY:

[No medium severity issues]

## LOW SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| L1 | `DepositToken::_transfer` is not protected against cross-contract reentrancy | **RESOLVED** |

In our previous audit, we pointed out that `DepositToken::_transfer` is not protected against reentrancy. In the latest version a `nonReentrant` guard was added; however this guard protects only against reentering the same method (which is the only `nonReentrant` method of `DepositToken`). Reentering from any method of the `Controller` is still possible, and although we could not find any concrete reentrancy attack, we believe that the lack of a cross-contract guard could potentially allow such attacks in future versions of the code.

For example, assume that some future functionality is added to `DepositToken::_beforeTokenTransfer` allowing code reentrancy (note that the method was empty in the initial version, but some code was added in the latest version). Then, imagine that an adversary has 20 ETH of deposit, of which 10 ETH is locked. The adversary can transfer his unlocked 10 ETH, and simultaneously withdraw the remaining locked 10 ETH, as follows:

1. Adversary calls `DepositToken::transfer` (for 10 ETH), which performs the `onlyIfNotLocked` check and then calls `_transfer`, which in turn calls `_beforeTokenTransfer`.
2. From `_beforeTokenTransfer`, the adversary reenters into Controller::withdraw (which has an independent nonReentrant guard), again for 10 ETH. The `DepositToken` state has not been updated yet, so `withdraw` will succeed (note also that `withdraw` calls `DepositToken::burnForWithdraw`, which however is *not* `nonReentrant`).

3. Returning to `_transfer`, the `onlyIfNotLocked` check has already been performed, so the method will happily transfer the requested 10 ETH (although the amount is now locked, after the withdrawal).

As a consequence, we recommend implementing a cross-contract reentrancy guard. Conceptually, a debt transfer is no different than a withdrawal (one can withdraw by transferring to an account without debt), so if we need to prevent two nested withdrawals, or two nested transfers, we also need to prevent a nester transfer+withdrawal.

## 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 | Gas optimization in DepositToken and DebtToken | **RESOLVED** |

`DepositToken::_beforeTokenTransfer` and `DebtToken::_beforeTokenTransfer` check the condition `balanceOf(_to) == 0` and if true call the controller to add the token to the list of tokens owned by `_to`. `DepositToken::_burn` calls `DepositToken::_beforeTokenTransfer` with the `_to` parameter set to `address(0)`, thus making an unnecessary call, which updates the DepositTokens owned by `address(0)`.

`DepositToken::_afterTokenTransfer` and `DebtToken::_afterTokenTransfer` exhibit a similar issue as there is no check that ensures the `_from` parameter cannot have the value `address(0)`. As a result, these methods unnecessarily spend gas to update the tokens lists of `address(0)` when they are called from `DepositToken::_mint` and `DebtToken::_mint respectively`.

| A2 | Inconsistent update requirements practice | **RESOLVED** |
|---|---|---|

Several update methods throughout the codebase, e.g., `Controller::updateOracle`, implement a require statement that ensures the new value is not equal to the one being replaced. However, this is not a consistent practice overall. `DefaultOracle::setPriceProvider` does not ensure that the new price provider is different from the current one, potentially leading to a “pointless” update. The same is true for the updates of the oracles and the default oracle in methods `MasterOracle::_updateOracles` and `MasterOracle::setDefaultOracle`. Also, several of the `Controller` methods responsible for updating values do not implement such requirements. The same applies to a few methods of `DepositToken` and `SynteticToken`.

| A3 | Duplicate check in `DefaultOracle::addOrUpdateAssetThatUsesChainlink` | **INFO** |
|---|---|---|

The statement `require(address(_asset) != address(0), "...");` can be removed from `DefaultOracle::addOrUpdateAssetThatUsesChainlink` as it’s also implemented in `DefaultOracle::_addOrUpdateAsset`, which the aforementioned method calls.

| A4 | Potentially misspelled method name | **RESOLVED** |
|---|---|---|

A misspelling might have occurred with the method name `addOrUpdated` of the `MasterOracle` contract, when the intended name would have been `addOrUpdate`.

| A5 | Unused `convert` methods | **RESOLVED** |
|---|---|---|

The use of the `convert` method in the various oracle contracts is confusing.

- `MasterOracle::convert` exists and is used, but internally uses convertToUsd/convertFromUsd.

- `DefaultOracle::convert` exists but does not seem to be used (note that IOracle::convert did exist in previous versions, but it was recently removed).
- The provider methods (`ChainlinkPriceProvider::convert`, `UniswapV3PriceProvider::convert`, …) do exist, but do not seem to be used.

One idea could be in `MasterOracle::convert` to use the underlying oracle's `convert` method, if both source/target assets have the same oracle. Otherwise all convert methods could be removed, if they are not used.

| A6 | `unchecked` optimization | **RESOLVED** |
|---|---|---|

The subtraction in `DepositToken::lockedBalanceOf` can be unchecked since `unlockedBalanceOf` can never exceed the account's balance. The gas savings will be small, but similar uses of `unchecked` do exist in the codebase, so there might be interest in such an optimization.

| A7 | Open issues from the previous audit | **INFO** |
|---|---|---|

The following issues (apart from L1 above) from the previous audit remain open. This is likely intentional, we just mention them for reference:

- H1 - `DebtToken::accrueInterest`'s algorithm is problematic
- M2 - Blocks-per-year constant seems off
- L2 - Contracts are difficult to initialize due to mutual dependency
- A3 - Some assets are assumed to be equivalent to USD
- A5 - Controller privileged functions can be blocked from making progress

| A8 | Compiler bugs | **INFO** |
|---|---|---|

Vesper Synth contracts were compiled with the Solidity compiler v0.8.9 which, at the time of writing, has no known issues.

## 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.