Vesper Synth audit

(+Vesper Pools delta)

Smart Contract Security Assessment

Jan 20, 2022

ABSTRACT

Dedaub was commissioned to perform a security audit of several smart contract modules of two independent Vesper protocols.

The scope of the audit was dual:

1. An audit of the new Vesper Synth protocol, performed on the contracts in the repository https://github.com/bloqpriv/vesper-synth, up to commit 143165906c47ce152826cfa3dcd028d0784b5b8d. A high-level description of the protocol was given in the following document.
2. Recent changes (on multiple files), of the Vesper Pools V3 protocol, from two PRs in the repository https://github.com/bloqpriv/vesper-pools-v3:
   - PR 843: from the previously audited commit 684b24469ff2252d3496c45c9f93aed9d3ad3ab6 up to the newer commit 5640f6583430203eafc8643c68f1c136b174fba9.
   - PR 849: from the commit 5640f6583430203eafc8643c68f1c136b174fba9 (of the PR 843 above) up to the newer commit fe609958b54b97463d958d14e5e8e3a8bdfdbe16.

Vesper Synth (part 1) had not been previously audited. Vesper Pools (part 2) had been previously audited (both v2 and v3), including the pools and several strategy contracts. The Vesper Pools delta of the current audit focuses mostly on the refactoring of Curve and Convex pool code.

The total auditing effort was 2 auditor-weeks, split in favor of Vesper Synth, since a new protocol has substantially higher chances to contain vulnerabilities. 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.

Concerning the Vesper Pools delta, note that an audit of small changes in a large protocol is necessarily out-of-context. We did a best effort to understand the changed lines of code and assess whether these changes are reasonable and do not introduce vulnerabilities. The audit, however, was restricted to the modified lines, and their interaction with the rest of the protocol is not always easy to assess.

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.

We particularly advise paying attention to the "Advisory" issues section of this audit report, which contains issues labeled just "Info". Although most of these are not code issues, they are protocol-wide and deployment-level considerations that should be kept in mind.

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:

ID	Description	STATUS
C1	`DepositToken::transferFrom` does not check the allowance	**OPEN**

`DepositToken::transferFrom` performs no check whatsoever that the sender has given allowance to the caller (it only checks that the sender's funds are not locked). As a consequence the contract is trivially vulnerable, allowing an adversary to steal an arbitrary amount of deposited funds, from both the protocol users as well as the treasury.

This can be easily fixed by adding the corresponding checks (which we assume were simply forgotten).

HIGH SEVERITY:

ID	Description	STATUS
H1	`DebtToken::accrueInterest`'s algorithm is problematic	**OPEN**

Interest compounding (the act of adding interest to the original amount, so that the interest itself is subject to further interest) is typically performed at regular, predefined intervals, e.g. once per year, or four times per year. This does not mean that the accounting has to be performed at exactly those intervals, it can certainly be performed more often or less often; however, the formula employed for compounding should ensure that independently from how often the accounting is performed, the final result is the same: updating the amount 10 times should produce the same amount as if the interest was calculated just once at the predefined interval.

However, the way `DebtToken::accrueInterest` is written, compounding happens at irregular intervals, depending on when the function happened to be called. If the function is called 10 times in some period of time, the interest calculated will be higher than it would be if the function was called once in the same period, leading to a non-deterministic behavior which will hinder the use of the protocol.

Specifically, the code does a linear computation of the current interest to accrue (just multiply a constant by the time elapsed) whereas the accrual between calls is compounding (i.e., exponential).

```
function accrueInterest() external override onlyController returns
(uint256 _interestAccumulated) {
      uint256 _currentBlockNumber = getBlockNumber();

      if (lastBlockAccrued == _currentBlockNumber) { return 0; }

      uint256 _blockDelta = _currentBlockNumber - lastBlockAccrued;
      uint256 _interestRateToAccrue =
            syntheticAsset.interestRatePerBlock() * _blockDelta;

      _interestAccumulated = _interestRateToAccrue.wadMul(totalSupply);
      totalSupply += _interestAccumulated;
      debtIndex += _interestRateToAccrue.wadMul(debtIndex);
}
```

We suggest to use the following standard formula for performing interest compounding **once per year**:

```
TS' = TS (1 + IR)^(BD / BPY)
where
TS  : original totalSupply
TS' : final totalSupply
IR  : annual interest rate
BD  : block delta since the last update
BPY : blocks per year
```

The formula has the property that performing an update twice at intervals BD1 and BD2, is equivalent to performing just once at interval BD1+BD2:

```
TS1 = TS  (1 + IR)^(BD1 / BPY)
TS2 = TS1 (1 + IR)^(BD2 / BPY)
    = TS  (1 + IR)^(BD1 / BPY) (1 + IR)^(BD2 / BPY)
    = TS  (1 + IR)^((BD1+BD2) / BPY)
```

If you prefer to compound N times per year, the formula can be changed to:

```
TS' = TS(1 + IR/N)^(N * BD / BPY)
where
N: number of compounds per year
```

Finally, could be added continuously (the limit of N going to infinity), using the following formula:

```
TS' = TS e^(IR * BD / BPY)
```

Both above formulas have the same property of being independent from the number of updates.

H2 Price oracle computation can overflow OPEN

[This issue is potentially high-severity, but we have not ascertained with high confidence. Raising it for the developers' review.]

In UniswapV2PriceProvider::_getAmountOut, the calculation seems to overflow easily.

```
function _getAmountOut(
      address _token,
      address _tokenIn,
      uint256 _amountIn
) private view returns (uint256 _amountOut) {
    PairOracleData memory _pairOracle = oracleDataOf[_token];
    if (_tokenIn == _pairOracle.token0) {
        _amountOut = _pairOracle.price0Average.mul(_amountIn).decode144();
    } else {
        require(_tokenIn == _pairOracle.token1, "invalid-token");
        _amountOut = _pairOracle.price1Average.mul(_amountIn).decode144();
    }
}
```

The two _amountOut computations multiply a fixpoint 112.112 bits quantity by an amount, which is scaled by e18 = ~60 bits. We believe the 112.112 fixpoint should be first decoded into a uint.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	The use of `type(uint256).max` to denote “all user’s balance” is risky, front-runnable	**OPEN**

In `Controller.sol`, `type(uint256).max` is used in several places to denote that the user wants to deposit/withdraw/mint/etc all funds in his balance. This functionality adds unnecessary risk of varying severity, depending on the semantics of each function.

The most important of all is the case of `Controller::repay`,when `_onBehalfOf != _sender`. In this case the sender requests a transaction asking to pay all debts of `_onBehalfOf`,for any amount (up to his own balance). A malicious beneficiary can observe this transaction, front-run it, and arbitrarily increase his own debt, leading to the victim repaying an arbitrarily larger amount than the one he intended to.

`Controller::deposit` poses a similar threat, albeit much harder to exploit. Again, assuming `_onBehalfOf != _sender,` the sender asks to deposit his entire balance to `_onBehalfOf.` It could happen that a malicious `_onBehalfOf` has an outstanding debt towards _sender in any ethereum protocol. By front-running the transaction, the adversary can repay his debt (which will register as paid in that protocol), causing the _sender's balance to increase, knowing that that entire balance will be transferred back to him by `Controller::deposit`.

We believe that this functionality is a "syntactic sugar" that does not justify the risk, and could be easily moved off-chain to the client-side. We recommend to remove it from all Controller functions.

M2	Blocks-per-year constant seems off	**OPEN**

The code in SyntheticAsset defines a BLOCKS_PER_YEAR constant:

```
uint256 public constant BLOCKS_PER_YEAR = 2102400;
// Dedaub: checked just now, got 2350729, over 10% difference
```

At audit time, this approximate number was significantly off, as our code comment illustrates.

LOW SEVERITY:

ID	Description	STATUS
L1	`DepositToken::_transfer` is not protected against reentrancy	**OPEN**

All core functions of the Controller contract (e.g. `Controller::withdraw`) are `nonReentrant`, to protect against reentrancy attacks. However, `DepositToken::_transfer` has no such guard, although transferring deposited funds is similar to withdrawing them.

Note that the risk of a reentrancy attack is particularly high for `DepositToken::_transfer`, since it checks not only the user's balance, but also that the funds are not locked. Balance checks are typically protected via Solidity's checked arithmetic: if one tries to withdraw more than his balance via a reentrancy attack, the balance would become negative and an underflow would occur. But such an underflow would not happen if an adversary tries to transfer funds that exist but are locked.

In the current code we could not find any concrete reentrancy attack; nevertheless we believe that the lack of a guard could potentially allow such attacks in future versions of the code.

For example, assume that some future functionality is added to `DepositToken::_beforeTokenTransfer` (which currently is empty) allowing code reentrancy. By reentering into `DepositToken::_transfer`, from within itself (after the `onlyIfNotLocked` check) an adversary could easily transfer an arbitrary amount of locked funds.

Similarly, the lack of guards allows the code to enter `DepositToken::_transfer` from within any Controller function. Again, we could not find a concrete attack, but we think it is likely that such an attack could be made possible in future versions of the code.

As a consequence, we recommend to treat `DepositToken::_transfer` in a similar way as the Controller functions, and implement a cross-contract reentrancy guard.

L2	Contracts are difficult to `initialize` due to mutual dependency	**OPEN**

The `DebtToken` and `SyntheticAsset` contracts mutually depend on each other, in the sense that both contracts' `initialize` function receives an instance of the other contract. Since anyone can call the `initialize` function, it should be called as early as possible; the OpenZepellin guidelines suggest calling `initialize` from the constructor of the Proxy contract (assuming that such a Proxy is used).

However, the mutual dependency makes it hard to follow this guideline, since one needs to create both contracts before initializing either of them, and only then call the `initialize` functions. Performing all tasks in a single transaction is non-trivial.

If the current design is maintained, it is likely that the contracts will be initialized in separate transactions. In such a case, we recommend paying particular attention and ensuring that all `initialize` calls succeed. If any such call reverts without being noticed, then the contracts could end up being initialized in an arbitrary way by a malicious adversary.

L3 Inconsistent upgrade check quantity OPEN

In `DebtTokenUpgrader::_checkResults`, the `decimals` quantity is uint8 but its result is read via `_checkUint256Results`.

```
function _calls() internal pure override returns (bytes[] memory calls) {
   calls = new bytes[](4);
   calls[0] = abi.encodeWithSignature("totalSupply()");
   calls[1] = abi.encodeWithSignature("decimals()");
   ...
}

function _checkResults(bytes[] memory _beforeResults, bytes[] memory
_afterResults) internal pure override {
    _checkUint256Results(_beforeResults, _afterResults, 0, 1);
    // checks totalSupply+decimals
    ...
}
```

We believe this works fine, but raise the issue of the inconsistency, in case it has not been considered.

OTHER/ ADVISORY ISSUES:

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

<table>
<tr><th>ID</th><th>Description</th><th>STATUS</th></tr>
<tr><td>A1</td><td>DepositToken::burnFromUnlocked is not used</td><td>OPEN</td></tr>
<tr><td colspan="3">DepositToken::burnFromUnlocked is marked as onlyController, but is not used by the Controller. We recommend removing obsolete functions.</td></tr>
<tr><td>A2</td><td>Multiple loops over all synthetic assets in the system</td><td>INFO</td></tr>
<tr><td colspan="3">The code contains multiple loops over all synthetic assets, e.g., in Controller:debtOf, Controller::depositOf. It seems that developers are aware of this, since it is mentioned in comments. However, we want to point out that even a loop that iterates over a couple of hundred storage elements may exceed the block gas limit. Any iteration over more than a few tens of storage locations is probably worth optimizing.</td></tr>
<tr><td>A3</td><td>Some assets are assumed to be equivalent to USD</td><td>INFO</td></tr>
<tr><td colspan="3">We are philosophically against assuming that a token is equivalent to USD and warn about it. Especially when there are on-chain and off-chain reliable pricing data, hard-coding a stable exchange rate of 1:1 for USD seems ill-advised, although the threat sounds exotic. The Oracle contract does exactly that for some assets, marked via flag isUsd. Trusting that some assets are USD-equivalent is the easiest way in which an issue affecting one token will become an ecosystem-wide issue. In terms of threats over top name “USD-equivalent” assets, we mention that, e.g., USDC is institutionally well-trusted, but, on-chain, its minting is ultimately governed by a single EOA; USDT is alleged to have systemic risks; TUSD is governed by a 2-of-4 multisig. Even if it’s unlikely that any of these assets fail to keep their peg, due to human error or financial threat, it will not be astonishing.</td></tr>
<tr><td>A4</td><td>Considerations regarding Uniswap v3 oracles</td><td>INFO</td></tr>
</table>

There are standard considerations relative to Uniswap v3 TWAP oracles that the developers are already aware of (e.g., ability to manipulate with limited liquidity). A lesser-known consideration is that Uniswap v3 pools shift to their clients the gas burden to expand the `observations` array of the pool. This means that for rare assets (which Vesper may want to look up in a Uniswap v3 TWAP) the pool needs its `observations` array expanded to enough positions to accommodate the pool's transactions that have occurred in the last TWAP period. This burden is upon the Vesper deployer. Expanding the `observations` array requires calling `increaseObservationCardinalityNext()` on the pool.

A secondary consideration is that the code currently uses a 3rd party non-upgradable contract for Uniswap V3 oracle computations:

```
/**
  * @notice The UniswapV3CrossPoolOracle contract address
  * @dev This is 3rd-party non-upgradable contract
  * https://etherscan.io/address/0x0f1f5a87f99f0918e6c81f16e59f3518698221ff#code
  */
IUniswapV3CrossPoolOracle public crossPoolOracle;
```

We only briefly reviewed this (out-of-scope) contract and it seems fine and analogous to computations we have seen elsewhere (e.g., in Chainlink contracts). Still, this is a point of caution to keep in mind.

A5	Controller privileged functions can be blocked from making progress	**INFO**

The code in `Controller::removeSyntheticAsset` and `removeDepositToken` has guards of the form:

```
require(_syntheticAsset.totalSupply() == 0, "supply-gt-0");
// Dedaub: but no straightforward way to make it zero
```

We remark that, although privileged, the governor of this contract has no way to ensure progress relative to this requirement. This is a fine design decision, just one to be aware of.

A6	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.

Vesper Pools contracts were compiled with the Solidity compiler v0.8.3 which, at the time of writing, has a known minor issue. We have reviewed the issue and do not believe it to affect the contracts. More specifically the known compiler bug associated with Solidity compiler v0.8.3 is:

- Memory layout corruption can happen when using abi.decode for the deserialization of two-dimensional arrays.

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.