Dedaub

Security Technology for Smart Contracts

# Vesper Finance

## Smart Contract Security Assessment

Date: September 6, 2021

## Abstract

Dedaub was commissioned to perform a security audit of several smart contract modules of the Vesper.finance protocol. We have previously audited a subset of the Vesper functionality but the newly audited code is either new or substantially rewritten.

The audit was performed on the contracts at (repo commit or archive title shared):

- https://github.com/vesperfi/vesper-pools-v3, commit `bdf1166bd0b2d0e55460a0921c3a68295998424a`

We have previously audited the Vesper v3 pools and several strategy contracts. The current audit focuses on newly introduced pools and strategies. Specifically, the scope of the audit is:

- The `CompoundXYStrategy` and `CompoundLeverageStrategy` strategies
- The Convex strategies
- The Alpha strategies
- The Rari Fuse strategies
- The Earn pool and strategies
- The VFR pools and strategies

Three auditors worked on the task over the course of one week. 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 audited code base is of substantial size, at around 3KLoC (excluding test and interface code). In addition, the audited code is part of a much larger system. Although we consulted all super-contracts (e.g., the base contracts for pools and strategies) and the code of external services (including Compound, Alpha, Convex), such code is outside the scope of this audit and is considered trusted. Mistakes of omission (e.g., methods that require overriding for correct functioning) are very hard to detect when auditing only sub-contracts added to an existing architecture. The development team should be aware of this limitation and test more thoroughly for conformance to expected protocols.

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.

There are expected centralization elements, as in most similar protocols. The "governor" of a pool has significant authority over user funds, therefore users implicitly trust the parties playing the governor role.

## Architecture and Recommendations

The overall architecture is elegant and largely event-based: a pool keeps track of strategies, sets constraints on the amounts of debt (i.e., investment) that a strategy is allowed to have, but otherwise all parties merely react to events: either periodic rebalances, or other environmental changes (e.g., change of fees, of debt limits, etc.). All event handling is coded defensively, handling different sets of conditions and reacting to conditions and not to causes. This is a strong architecture for growth.

We have made some recommendations in our previous audits regarding better documentation along several axes. These continue to apply. Also, as in earlier audits, we recommend that the swapping functionality be reconsidered, so that it receives objective price oracle information and uses it to establish a minimum returned amount per swap.

A recommendation from the current audit, with the CompoundXYStrategy as example, is to add a last-resort emergency rescue facility for tokens that might otherwise be stuck in a strategy. In particular, it is unclear whether the rate of real-world updates permits scenarios such as:

- The borrowed token from Compound rises dramatically in value compared to the collateral token.
- The loan becomes undercollateralized and is subject to liquidation calls, by anyone. The strategy does not move quickly enough to change this in any way.
- The strategy is left with high-value borrow tokens that it can neither move nor exchange.

The above is a specific example, but the point is that a general token rescue facility can be used for any such scenario that results from circumstances outside the parameters foreseen in the code.

We list specific important issues in the next section.

## Vulnerabilities and 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>1) User or system funds can be lost when third party systems misbehave.<br>2) DoS, under specific conditions.<br>3) Part of the functionality becomes unusable due to programming error. |
| **Low** | Examples:<br>1) Breaking important system invariants, but without apparent consequences.<br>2) Buggy functionality for trusted users where a workaround exists.<br>3) 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

| Nr. | Description | Status |
|---|---|---|
| **H1** | **Swapping (or consulting an oracle for pricing) can be front-run** | **Open** |

There are many instances of Uniswap/Sushiswap swaps and oracle queries (mainly wrapped in calls to the internal function `Strategy::_safeSwap`, but also as direct calls to `swapTokensForExactTokens`, and calls to `swapManager.safeGetAmountsOut`, `bestOutputFixedInput`) that can be front-run or return biased results through tilted exchange pools. Fixing this requires careful thought, but the codebase has already started integrating a simple time-weighted average price oracle.

We have warned about such swaps in past audits and the saving grace has been that the swapped amounts are small: typically interest/reward payments only. Thus, tilting the exchange pool is not profitable for an attacker. In CompoundXYStrategy (which contains many of these calls), swaps are performed not just from the COMP rewards token but also from the collateral token. Similarly, in the Earn strategies, the `_convertCollateralToDrip` does an unrestricted collateral swap, on the default path (no `swapSlippage` defined).

Swapping collateral (up to all available) should be fine if the only collateral token amounts held in the strategy at the time of the swap are from exchanging COMP or other rewards. Still, this seems like a dangerous practice.

Standard background: The problem is that the swap can be sandwiched by an attacker collaborating with a miner. This is a very common pattern in recent months, with MEV (Maximum Extractable Value) attacks for total sums in the hundreds of millions. The current code complexity offers some small protection: typically attackers colluding with miners currently only attack the simplest, lowest-risk (to them) transactions. However, with small code analysis of the Vesper code, an attacker can recognize quickly the potential for sandwiching and issue an attack, first tilting the swap pool and then restoring it, to retrieve most of the funds swapped by the Vesper code.

In the current state of the code, the attacker will likely need to tilt two pools: both Uniswap and Sushiswap. However, this also offers little protection, since they both use similar on-chain price computations and near-identical APIs.

In the short-term, deployed code should be closely monitored to ensure the swapped amounts are very small (under 0.3%) relative to the size of the pools involved. Also, if an attack is detected, the contract should be paused to avoid repeat attacks.

However, the code should evolve to have an estimate of asset pricing at the earliest possible time! This can be achieved by using the TWAP functionality that is already being added, with some tolerance based on this expected price.

| | | |
|---|---|---|
| **H2** | **VFRBuffer: non-standard ERC20 Tokens can be stuck inside the VFRBuffer** | **Resolved** |

| |
|---|
| The VFRBuffer does not use the safeERC20 library for the transfer of ERC20 tokens. This can cause non-standard tokens (for example USDT) to be unable to be transferred inside the Buffer and get stuck there. This issue would normally be ranked lower, but since USDT is actively used in past strategies, it seems likely to arise with upcoming instantiations of the VFR pool. |

## Medium Severity

| Nr. | Description | Status |
|---|---|---|
| **M1** | **COMP rewards might get stuck in CompoundLeverageStrategy** | **Dismissed (Normal path: rebalance before migrate)** |

CompoundLeverageStrategy does not offer a way to migrate COMP tokens that might have been left unclaimed by the strategy up to the point of migration. What is more, COMP is declared a reserved token by CompoundMakerStrategy making it impossible to sweep the strategy's COMP balance even if a claim is made to Compound after the migration. The `_beforeMigration` hook should be extended to account for the claim and consequent transfer of COMP tokens to the new strategy as follows:

```
function _beforeMigration(address _newStrategy) internal virtual override {
   require(IStrategy(_newStrategy).token() == address(cToken),
"wrong-receipt-token");
   minBorrowLimit = 0;
   // It will calculate amount to repay based on borrow limit and payback all
   _reinvest();
   // Dedaub: Claim COMP and transfer to new strategy.
   _claimComp();
   IERC20(COMP).safeTransfer(_newStrategy,IERC20(COMP).balanceOf(address(this)));
}
```

| Nr. | Description | Status |
|---|---|---|
| **M2** | **COMP rewards might get stuck in CompoundXYStrategy** | **Dismissed (as above)** |

The `_beforeMigration` hook of CompoundXYStrategy calls `_repay` and lets it handle the claim of COMP and its conversion to collateral, thus no COMP needs to be transferred to the new strategy prior to migration. However, the claim in `_repay` happens only when the condition `_repayAmount > _borrowBalanceHere` evaluates to true, which might not always hold prior to migration, leading to COMP getting stuck in the strategy. This is because COMP is declared a reserved token and thus cannot be swept after migration.

| | | |
|---|---|---|
| **M3** | **CompoundLeverageStrategy: Compound markets are never entered** | **Dismissed (unnecessary)** |

The CompoundLeverageStrategy's CToken market is never entered via Compound's Comptroller. This leaves the strategy unable to borrow from the specified CToken.

| | | |
|---|---|---|
| **M4** | **VFRStablePool: The checkpoint method only considers profiting strategies when computing the total profits of a pool's strategies** | **Resolved** |

The `checkpoint()` method of the VFRStablePool iterates over the pool's strategies to compute their total profits and update the pool's `predictedAPY` state variable:

```
address[] memory strategies = getStrategies();
uint256 profits;
// SL: Is it ok that it doesn't consider strategies at a loss?
for (uint256 i = 0; i < strategies.length; i++) {
  (, uint256 fee, , , uint256 totalDebt, , , ) =
IPoolAccountant(poolAccountant).strategy(strategies[i]);
  uint256 totalValue = IStrategy(strategies[i]).totalValueCurrent();
  if (totalValue > totalDebt) {
    uint256 totalProfits = totalValue - totalDebt;
    uint256 actualProfits = totalProfits - ((totalProfits * fee) /
MAX_BPS);
    profits += actualProfits;
  }
}
```

The above computation disregards the losses of any strategies that are not profiting. Due to that the predicted APY value will not be accurate.

| M5 | The CompoundXY strategy does not account for rapid rise of borrow token price | Resolved |
|---|---|---|
| (This issue was also used earlier as an example in our architectural recommendations.)<br><br>The CompoundXY strategy seeks to repay a borrowed amount if its value rises more than expected. However, volatile assets can rise or drop in price dramatically. (E.g., a collateral stablecoin can lose its peg, or a token's price can double in hours.) This means that the Compound loan may become undercollateralized. In this case, the borrowed amount may be worth more than the collateral, so it would be beneficial for the strategy to not repay the loan. Furthermore, it might be the case that the collateral gets liquidated before the strategy rebalances. In this case the strategy will be left with borrow tokens that it can neither transfer nor swap.<br><br>The strategy can be enhanced to account for the first of these cases, and the overall architecture can adopt an emergency rescue mechanism for possibly stuck funds. This emergency rescue would be a centralization element, so it should only be authorized by governance. | | |

| M6 | CompoundXYStrategy, CompoundLeverageStrategy: Error code of Compound API calls ignored, can lead to silent failure of functionality | Mostly Resolved |
|---|---|---|
| The calls to many state-altering Compound API calls return an error code, with a 0-value indicating success. These error codes are often ignored, which can cause certain parts of the strategies functionality to fail, silently.<br>The calls with their error status ignored are:<br>● `CompoundXYStrategy::constructor: Comptroller.enterMarkets()`<br>● `CompoundXYStrategy::updateBorrowCToken: Comptroller.exitMarket(), Comptroller.enterMarkets(), CToken.borrow()`<br>● `CompoundXYStrategy::_mint: CToken.mint()` (is returned but not check by the callers of `_mint()`)<br>● `CompoundXYStrategy::_reinvest: CToken.borrow()`<br>● `CompoundXYStrategy::_repay: CToken.repayBorrow()`<br>● `CompoundXYStrategy::_withdrawHere: CToken.redeem(), CToken.redeemUnderlying()`<br>● `CompoundLeverageStrategy::_mint: CToken.mint()`<br>● `CompoundLeverageStrategy::_redeemUnderlying: CToken.redeemUnderlying()`<br>● `CompoundLeverageStrategy::_borrowCollateral: CToken.borrow()`<br>● `CompoundLeverageStrategy::_repayBorrow: CToken.repayBorrow()` | | |

## Low Severity

| Nr. | Description | Status |
|---|---|---|
| **L1** | **AlphaLendStrategy: ALPHA rewards are not claimed on-chain** | **Open** |
| The `_claimRewardsAndConvertTo()` method of the Alpha lend strategy does not do what its name and comments indicate it does. It only converts the claimed ALPHA tokens. The actual claiming of the funds does not appear to happen using an on-chain API. | | |
| **L2** | **Unused storage field** | **Resolved** |
| In CompoundLeverageStrategy, field `borrowToken` is unused. A comment mentions it but does not match the code. | | |
| **L3** | **Two swaps could be made one, for fee savings** | **Dismissed, detailed consideration** |
| In `CompoundXYStrategy::_repay`, COMP is first swapped into collateral, and then collateral (which should be primarily, if not exclusively, the swapped COMP) is swapped to the borrow token. This incurs double swap fees. | | |

## Other/Advisory Issues

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

| Nr. | Description | Status |
|---|---|---|
| **A1** | **VFRCoveragePool contract seems to serve no purpose** | **Open** |
| This contract currently does nearly nothing. It is neither inherited nor exports functionality that makes it usable as part of a VFR strategy. | | |

| | | |
|---|---|---|
| **A2** | **VFR contract is there only for code reuse** | **Open** |

The VFR contract currently has the form:

```
abstract contract VFR {
 function _transferProfit(...) internal virtual returns (uint256) {...}
 function _handleStableProfit(...) internal returns (uint256 _profit) {...}
 function _handleCoverageProfit(...) internal returns (uint256 _profit) {...}
}
```

It is, thus, a contract that merely defines internal functions, used via inheritance, for code reuse purposes. "Inheritance for code reuse" is often considered a bad, low-level coding practice. A similar effect may be more cleanly achieved via use of a library.

| | | |
|---|---|---|
| **A3** | **Inconsistent "reserved" tokens** | **Open** |

In most strategies the collateral token is part of those in `isReservedToken`. Not in AlphaLendStrategy.

| | | |
|---|---|---|
| **A4** | **Claiming COMP rewards can be triggered by anyone** | **Dismissed, after review** |

Although we cannot see an issue with it, multiple public functions allow anyone to trigger a claim of COMP rewards, e.g., in CompoundLeverageStrategy methods `totalValueCurrent`/`isLossMaking`, and similarly in CompoundXYStrategy. It is worth revisiting whether the timing of rewards can confer a benefit to a user.

| | | |
|---|---|---|
| **A5** | **Inconsistent conventions** | **Resolved** |

Similar functionality often follows different conventions. For instance, between CompoundXYStrategyETH and CompoundLeverageStrategyETH, we notice a difference in the `_mint` function (in one case it returns a value in the other not), and the presence of an `_afterRedeem` vs. full overriding of `_redeemUnderlying`.

| | | |
|---|---|---|
| **A6** | **RariFuseStrategy: looser checks are performed on construction than on migrateFusePool()** | **Resolved** |

When the RariFuseStrategy is constructed, a CToken (assumed to belong to an instantiation of a Rari Fuse pool) is passed as an argument. However, when the strategy migrates to another Fuse pool, Fuse's API is used to ensure the new CToken will be part of a Rari Fuse pool. The same checks should also take place during the contract's construction.

| A7 | Compiler bugs | Info |
|---|---|---|
| The 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:` <br><br> ● 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 status 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](contract-library.com) service, which decompiles and performs security analyses on the full Ethereum blockchain.