Dedaub

Security Technology for Smart Contracts

Vesper Finance Pools

Smart Contract Security Assessment

Date: May 9, 2021

Abstract

Dedaub was commissioned to perform a security audit of the Vesper.finance pools smart contracts, version 3. We have previously audited the Vesper pool and strategy contracts, however there have been significant functionality changes in the new version: multi-strategy pools, removal of staking, removal of controller, new interaction with strategies.

The audit was performed on the contracts at https://github.com/vesperfi/vesper-pools-v3, commit `021794c4aa4247f995ad79d62d52c304bea3bd71`. This was an audit with a short deadline and focused on a subset of the functionality, intended for urgent release. Specifically, the audited parts contain:

- the new pool architecture (PoolShareToken, VTokenBase, the specific instantiation VUSDC);
- the Aave strategy (Strategy, AaveCore, AaveStrategy, AaveStrategyUSDC);
- supporting functionality (Governed, Owned, Pausable, UniswapManager).

Two 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 modest size, at around 1.5KLoC (excluding test and interface code).

The audit focused on security, establishing the overall security model and its robustness, and also crypto-economic issues. Functional correctness (e.g., that the calculations are correct) was a secondary priority. 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.

With such a reactive, asynchronous, event-based architecture, however, we need to emphasize the need for excellent documentation, both in the code and possibly in accompanying documents. Such documentation should make clear:

- What states a strategy or pool can be in and under what conditions;
- What invariants are maintained every time a pool makes changes to a strategy's state and vice versa. This should include accounting invariants (e.g., that the pool's accounting for total funds per strategy is accurate up to the last rebalance of a strategy) as well as interaction invariants (e.g., that the strategy is responsible for calling `reportEarning` with the `payback` amount satisfying certain conditions). Notably, since the computation of current funds in a pool does not synchronize all strategies, the amount computed could be one that has never been "actual funds" for any past moment in time: every strategy's funds could have been fluctuating, with `rebalance` called at different times per strategy.
- What is the logic for transfer and management of funds (e.g., that a strategy can only draw funds from the pool upon a rebalance, that it should return gains to the pool as upon a rebalance, for reinvestment, that it should liquidate to cover losses at the point of a rebalance).

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

[No high severity issues]

Medium Severity

Nr.	Description	Status
1	**AAVE staking rewards cannot be claimed**	**Resolved**
Aave staking rewards are claimable by calling the `claimRewards()` function of the Staked AAVE (stkAAVE) contract. This functionality is invoked in `AaveCore::_claimAave()`, used in `AaveStrategy::_claimRewardsAndConvertTo()` function which is however not reachable from any of the external entry points of AaveStrategy. This makes the strategy unable to claim any of its staking rewards.		
2	**VTokenBase: inconsistent state of `withdrawQueue` storage variable**	**Resolved**

The `address[] public withdrawQueue` state variable of VTokenBase has consistency issues.

- When the governor sets a new value for the `withdrawQueue` in `VTokenBase::updateWithdrawQueue()` the queue does not have to include all the currently active strategies.
- Furthermore when a new strategy is added to the vPool in `VTokenBase::addStrategy()` it is not added to the queue.

Even though the former is acknowledged in the comments, the latter and more likely to happen (by the governor adding a new strategy and forgetting to update the `withdrawQueue`) is not.

If under any of the above an active strategy is not part of the `withdrawQueue` but holds funds, it can lead to a user attempting to withdraw collateral from the pool to obtain less funds due to `VTokenBase::_withdrawCollateral()` not being able to recover the funds of all the strategies.

3	**Strategy activation model unclear/loose**	**Resolved(?)**

The semantics of the `active` field of `StrategyConfig` are unclear from the code. In `VTokenBase::addStrategy()` the value of `active` for the new strategy is passed as a parameter, allowing for the introduction of new strategies that are "inactive". However the contract offers no easy way to activate it after its addition.

It seems that the only way to activate a strategy that was added as "inactive" is to re-add it via `VTokenBase::addStrategy` (which will require in a second `push` to the strategies array), or by calling `VTokenBase::migrateStrategy()` with both `_old`, and `_new` argument values pointing to the same strategy. We do not know if this intentional or due to a missing check, enforcing that `(_old != _new)`.

Nevertheless, this behavior should be defined more clearly and documented. If `addStrategy` is to be always followed by a `migrateStrategy`, this should be documented. If the old strategy in a `migrateStrategy` has to be inactive, this should be enforced in the code. Such a protocol would also cover the earlier issue (inconsistent state of `withdrawQueue`). Perhaps a strategy should be added to the `withdrawQueue` always at the same time as becoming active?

Low Severity

Nr.	Description	Status
1	**Unsafe calls to ERC20 functions**	**Resolved**

In `VTokenBase::reportEarning()` and `VTokenBase::sweepERC20()` calls to ERC20 functions `transfer()` and `transferFrom()` are made not using the OZ SafeERC20 wrappers. This makes them incompatible with non-standard tokens that are deployed.

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
1	**VTokenBase: inconsistent state of `strategies` storage variable**	**Resolved**
The `address[] public strategies` state variable of VTokenBase has consistency issues. Strategies are only added to this array and never deleted. This finding is acknowledged in the contract's code, and is minor because this variable is currently never used.		
2	**Strategy's active status could be boolean?**	**Resolved**
The `active` field of the StrategyConfig structure is only used in checks against zero and assignments of 0/1. It is not clear whether more than a boolean will be necessary in the future.		
3	**Inconsistent code**	**Resolved**

In the `VTokenBase::onlyStrategy()` modifier `msg.sender` is used instead of the `_msgSender()` wrapper used throughout the contract:
`modifier onlyStrategy() {`
`  require(strategy[msg.sender].active != 0,`
`  "caller-is-not-active-strategy");`
`  _;`
`}`		

4	Redundant check	Resolved

In the `VTokenBase::reportEarning()` the first `require` statement repeats the check made by the `onlyStrategy` modifier:

```
function reportEarning(
  uint256 _profit,
  uint256 _loss,
  uint256 _payback
) external onlyStrategy {
  require(strategy[_msgSender()].active != 0, "strategy-not-active");
...
```

5	Dead code	Optimization

In `Strategy::createGuardianList()` the following code is dead, under the current `onlyGovernor` modifier:

```
if (_msgSender() != _governor) {
  require(guardians.add(_msgSender()), "add-guardian-failed");
}
```

6	Comment inconsistent with actual invariant	Resolved

In `Strategy::withdrawAll`, the comment states an invariant that is not fully accurate:

```
//Pool has a require check on (payback + profit = collateral in strategy)
```

The actual check is "`payback + profit <= collateral in strategy`".

7	Hard-coded factory contract address	Info

`Strategy::createGuardianList` contains a hard-coded address (of the factory contracts). We recommend lifting this to (at least) a constant at the top of the contract, if not a deployment configuration parameter.

8	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`: • 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 service, which decompiles and performs security analyses on the full Ethereum blockchain.