Dedaub

Security Technology for Smart Contracts

Vesper Finance Pools

Smart Contract Security Assessment

Date: Apr. 30, 2021

Abstract

Dedaub was commissioned to perform a security audit of the Vesper.finance pools smart contracts.

The audit was performed on the contracts available on https://github.com/vesperfi/vesper-pools at commit `89594d687dae11d4f5c54cca6b3277e6fc72a5be`.

Four auditors worked on the task over the course of two weeks. 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 code base is large, at over 7KLoC, or around 6KLoC without interface code. There is significant replication of code throughout the strategies, however.

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 “owner” of the Controller contract (currently a 3-of-5 multisig) has significant authority over user funds, therefore users implicitly trust the parties playing the owner role. Attack-by-owner vectors have not been considered in this audit, but it is clear that the effects can be dramatic (e.g., the owner can change registered contracts, such as strategies or the Uniswap router, and drain funds).

Architecture and Recommendations

The code consists of two main groups of functionality: (growth) pools and strategies to manage them. Pools and strategies are accompanied by governance, staking, and management components.

Most interesting elements (pool shares, stakes, rewards) are tokenized, by reuse of OpenZeppelin (OZ) libraries.

The governance and staking functionalities are both standard code, employed in several similar projects. They both leverage ERC-20 hooks (e.g., the `_beforeTokenTransfer` hook of the OZ ERC-20 implementation) to intervene uniformly at appropriate points.

Our high-level architectural recommendations are:

- To greatly reduce or eliminate code duplication. The current strategies often implement the same features multiple times in identical or near-identical ways. This is highly error-prone, as inconsistencies are very easy to introduce. The code does not employ libraries anywhere--the only code reuse is through inheritance. Libraries could alleviate many of the duplication problems and are recommended.
- To document interaction protocols clearly, especially those of an event-based nature. This includes all the hooks for actions. It is currently hard to know what hooks every contract has to override to function correctly. Examples of current complications:
    - VVSP should support both (Synthetix-style) staking and governance (vote delegation) functionality. (The documentation states that VVSP staking is an upcoming feature.) Both of these frameworks tokenize their functionality and require actions in the `_beforeTokenTransfer` hook. However, only governance functionality is there at the moment. VVSP staking is currently incorrect: any incoming transfer of the staking token would not call `updateReward` before and would then collect all rewards since the beginning of time (and no further updates later).
    - In PoolShareToken, getting a deposit from the user before minting is left to the `_beforeMinting` hook, which, however, has an empty implementation. The sub-contracts are supposed to override it. If they forget, a deposit will mint the caller new tokens without ever receiving the caller's investment.
    - VSP currently supports both staking and vote delegation. But it does so by sprinkling `_moveDelegates` calls to all functions that move tokens (`transfer`, `transferFrom`, `mint`, `burn`, `burnFrom`). Treating this uniformly is exactly the purpose of the `_beforeTokenTransfer` hook, which is currently used only for staking functionality.

  A clearly-documented protocol for which hooks to implement and for what purpose (staking, governance) will be important for avoiding future bugs.
- To document the possible states of investments clearly. For instance for strategies combining Maker with another service, the documentation can distinguish:

- Underwater: there is more Maker debt than the DAI investment value. When does this happen? Only when investments drop in value?
- Collateral ratio below low water: the Maker debt needs to be reduced. When does this happen? Only when collateral in Maker loses value?
- Collateral ratio between low water and high water.
- Collateral ratio above high water: too much collateral. Only happens because investments gave back interest that became collateral?
- Value of collateral is less than the Maker "minimum Debt" times highWater. Does this happen only in extremely low-collateralization pools?

  What are the allowed combinations of the above?

We believe that the above kinds of architectural or documentation improvements will be the best investment in the protocol's stability and security. Each pool and each strategy should have clear requirements as to what it needs to implement and should not re-implement it if an identical implementation is already employed elsewhere.

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

Description	Status
The `resurface` function in the Maker, AaveMaker, and AaveV2Maker strategies performs a Uniswap swap of potentially unlimited impact. This allows an attacker to steal the entire collateral balance of the pool. (Whether this balance is high or regularly re-committed to the Maker loan is an orthogonal issue, but may limit the impact of such an attack.) Specifically, the code performs a swap as follows: `uniswapRouter.swapExactTokensForTokens(tokenNeeded, 1, path, address(this), now + 30);` The `tokenNeeded` amount, however, is also obtained from Uniswap, based on the current state of the Uniswap liquidity pool: `uint256 tokenNeeded = uniswapRouter.getAmountsIn(shortAmount, path)[0];` Effectively, the code swaps “as much collateral token as needed” to cover the `shortAmount`. The attacker can perform a sandwich attack: they can tilt the Uniswap liquidity pool with a flash loan so that the collateral appears worthless. The subsequent call to `resurface` (assuming the pool is underwater) will cause the Vesper pool to spend all its collateral tokens. The attacker then performs an inverse swap to bring the Uniswap liquidity pool back to balance. The result of the second swap is that the attacker gets all the DAI that the Vesper pool lost.	**Resolved**

High Severity

Description	Status
The `_rebalanceEarned` function in abstract contracts `MakerStrategy`, `CompoundStrategy` and contracts `AaveMakerStrategy`, `AaveV2MakerStrategy`, performs token swaps that are vulnerable to sandwich attacks. In the Maker-related strategies `_rebalanceEarned` is responsible for swapping-and-transfer excessive DAI tokens from the Maker Vault to the Grow Pool as collateralToken and is called by the following functions: `function rebalance() external override live {` `  _rebalanceEarned();` `  _rebalanceCollateral();` `}` `function rebalanceEarned() external live {` `  _rebalanceEarned();` `}` Both `rebalance` and `rebalanceEarned` are external functions that can be called by anyone as long as the strategy is not paused, leaving space for a sandwich attack where an adversary may first tilt the uniswap pool of DAI-*collateralToken* and then calling one of the above functions resulting in a bad swap. In a similar way, an adversary could attack the Compound strategies. In these strategies `_rebalanceEarned` claims the COMP tokens accrued as interest and, after swapping for *collateralToken* of the associated Grow Pool, transfers them to the Grow Pool. It is called by function	**Resolved**

`` function rebalance() external override live { _rebalanceEarned(); uint256 balance = collateralToken.balanceOf(pool); if (balance != 0) { _deposit(balance); } } ``` In this case the attack involves tilting the uniswap pool COMP-*collateralToken.*	
AaveStrategy may be attacked so that it cannot collect any fees upon the accrued interest. The strategy intends to keep track of the interest accrued so as to distribute 15% as platform fees. Towards this end, at every `withdraw` and `deposit` the amount of fees to-be-collected is updated by calling the function `updatePendingFee`. ``` function _calculatePendingFee(uint256 aTokenBalance) internal view returns (uint256) { uint256 interest = aTokenBalance.sub(aToken.principalBalanceOf(pool)); uint256 fee = interest.mul(controller.interestFee(pool)).div(1e18); return pendingFee.add(fee); } function _updatePendingFee() internal { pendingFee =_calculatePendingFee(aToken.balanceOf(pool)); } ``` `_calculatePendingFee` relies on Aave code for keeping track of the pool's aTokens *principal balance,* i.e. the balance with the interest accrued subtracted. The principal balance is updated on each aToken's transfer from/to the pool, so an attacker could make all of the pool's balance to always be counted as principal by sending tiny amounts to the pool's balance.	**Wont fix(Strategy deprecated)**

Medium Severity

Description	Status
A sandwich attack similar to the above attacks (to `rebalance` or `resurface`) can be performed against the `sweepErc20` function of vTokens. The code (in `VTokenBase::_sweepErc20`) performs a Uniswap swap from the swept token to the underlying token. It seems that this is lower severity than prior attacks because such tokens are mostly "dust" and not accumulated in large quantities during normal operations.	**Wont fix**
The controller doesn't offer a way to atomically upgrade the strategy of a pool. This is due to the fact that the `updateStrategy()` function requires the current strategy to be upgradable: `if (currentStrategy != address(0)) {` `  require(IStrategy(currentStrategy).isUpgradable(),` `  "strategy-is-not-upgradable");` `  vpool.resetApproval();` `}` The `isUpgradable()` check ensures that the strategy to be removed has no funds locked into it. For it to succeed the controller would normally have to previously have withdrawn all the funds from the Strategy (normally by calling `withdrawAll()`). The way the controller is structured does not allow the withdrawal of the pool's funds and the call to `Controller::updateStrategy()` to be performed in the same transaction. This allows an external actor to DoS a strategy upgrade by causing a transfer of funds to the old strategy between the withdrawal of its funds by the controller and the call to `updateStrategy()`. This transfer of funds can either be performed an attacker sending a negligible amount of aTokens/cTokens in `Aave{V2}Strategy/CompoundStrategy` or by	**Open**

calling either of the publicly reachable `rebalanceCollateral()` or `deposit()` functions for Maker-based strategies. In the latter case this attack can be mitigated by the controller calling `vpool.resetApproval()` in the same transaction as `withdrawAll()` using the `executeTransactions()` functionality. The former case, however, cannot currently be mitigated by means within the power of the controller.	

Low Severity

Description	Status
In `AaveStrategy::sweepErc20` there is logic for transferring loose ETH to the pool. There doesn't seem to be any way to salvage ETH from the pool (although the interface `IVesperPool` defines function `withdrawETH`, the pools do not implement it). Only the vETH pool has a way to withdraw ETH (and even this seems to be up to amounts proportional to shares, so some ETH will be left over forever, if it is ever dropped into the strategy and swept into the pool).	**Wont fix (Strategy deprecated)**

Other/Advisory Issues

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

Description	Status
Function `Controller::updateInterestFee()` lacks the `validPool` modifier similar functions employ. While this is still protected functionality we suggest it be used for consistency.	**Suggestion**

The separation of the addition of a new pool in `Controller::addPool()` from the setting of its parameters, leaves room for the misconfiguration of pools. While for most parameters this cannot lead to undefined behavior, the fee parameters could be left to 0 by mistake, without that being intended. We suggest that a pool's parameters should be set upon its addition by the Controller.	**Suggestion**
Removing a pool in `Controller::removePool()` does not offer a way to clear the pool's parameters from the contract's state. We suggest for them to be cleared upon the pool's removal to reduce state clutter and benefit from gas refunds.	**Suggestion**
In contract `PoolShareToken.sol`, function: `function _deposit(uint256 amount) internal whenNotPaused` may be called by one of the functions: `function deposit(uint256 amount) external virtual nonReentrant whenNotPaused` `function depositWithPermit( uint256 amount, uint256 deadline, uint8 v, bytes32 r, bytes32 s ) external virtual nonReentrant whenNotPaused` Modifier `whenNotPaused` is redundant for the internal function `_deposit()` and is suggested to be removed in order to save gas.	**Suggestion**

Code such as the example below appears in VVSP, VtokenBase, AaveMakerStrategy, AaveV2MakerStrategy, CompoundStrategy, MakerStrategy, VSPStrategy. `uniswapRouter.swapExactTokensForTokens(amt, 1, path, address(this), now + 30);` The `now+30` expression is extraneous. The swap completes synchronously, within the same transaction. The expiration parameter only makes sense with an absolute number. Otherwise `now` serves the same purpose as `now+30`.	**Suggestion**
There are a number of redundant type casts in implementations of function: `function sweepErc20(address _erc20) external` This is a member function of strategies and pools responsible for handling non-native ERC20 tokens, typically by swap-and-deposit them back into the corresponding Grow Pool. The redundant type casts are the following. In contract VVSP.sol `function sweepErc20(address _erc20) external {` `  // [...]` `  address[] memory path;` `  // dedaub: redundant cast to address` `  if (address(_erc20) == WETH) {` `  path = new address[](2);` `  // dedaub: redundant cast to address` `  path[0] = address(_erc20);` `  path[1] = address(token);` `  } else {` `  path = new address[](3);` `  // dedaub: redundant cast to address`	**Suggestion**

```
            path[0] = address(_erc20);
            // dedaub: redundant cast to address
            path[1] = address(WETH);
            path[2] = address(token);
      }
      // [...]
}
```

Also, in contract VSPStrategy.sol

```
function _rebalanceEarned(IVesperPool _poolToken, uint256
_amt) internal {
      // [...]
      if (address(from) == WETH) {
            path = new address[](2);
            path[0] = address(from);
            path[1] = address(vsp);
      } else {
            path = new address[](3);
            path[0] = address(from);
            // dedaub: unnecessary cast to address
            path[1] = address(WETH);
            path[2] = address(vsp);
      }
      // [...]

}
```

More redundant type casts may exist in the code base and are harmless, but could be removed for clarity.

Events are generally describing what call was made not what action was performed. This is a fine design decision, as long as external mechanisms do not trust these events to update state. It is a better practice to list what was actually performed, or *both* the request and the action performed. For instance, `_afterBurning` (in all vTokens) may transfer less than the amount requested, if not enough collateral can be withdrawn. However, the emitted event reflects the amount requested, not that transferred.	**Suggestion**
The contracts were compiled with the Solidity compiler `v0.6.12` which, at the time of writing, has known minor issues. We have reviewed the issues and do not believe them to affect the contract. More specifically below are the known compiler bugs associated with Solidity compiler `v0.6.12`: • Wrong (cached) value for consecutive keccak hashes produced in inline assembly, hashing memory contents starting from the same index but with different lengths. • Memory layout corruption can happen when using abi.decode for the deserialization of two-dimensional arrays. • Copying an empty `bytes` or `string` array from memory to storage can cause data corruption: • Direct assignments of storage arrays with an element size <= 16 bytes (more than one values fit in one 32 byte word) are not correctly cleared if the length of the newly assigned value is smaller than the length of the previous one. (No such array is ever stored.)	**Info**

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.