Vesper Protocol audit

Smart Contract Security Assessment

Oct 12, 2021

ABSTRACT

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

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

- https://github.com/bloqpriv/vesper-pools-v3, commit 57f4e471e7d5afc64e96d2dce717aaecb2d53515, continuing our previous audit, which had considered changes up to commit bdf1166bd0b2d0e55460a0921c3a68295998424a
- https://github.com/bloqpriv/vesper-pools, from commit df525ce011dfbf4fedaffb213d164b4aa52a1c34 to 8db2c976ebf340ae069c83ad2de6574037b2032f
- pf-payment-stream-master-Aug31 (received as zip archive).

We have previously audited the Vesper v2 and v3 pools and several strategy contracts. The current audit focuses mostly on verifying newly introduced changes to existing pools and strategies. Specifically, the audit scope includes:

- The PaymentStream and PaymentStreamFactory contracts
- The newly added Aave and dYdX flash loan functionality
- The PoolRewards and PoolRewardsUpgrader contracts
- The VFR pools and strategies
- The Earn pool and strategies
- The CompoundXY, CompoundLeverage and other Compound strategies
- The Maker strategies
- The Alpha strategies
- The Rari Fuse strategies

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 consists of some-3000 LoC of diffs in the Vesper pools (primarily v3, with about 300 lines in v2) and some-400 LoC in the payment stream module. Auditing diffs may miss issues that require a full understanding of context. Given the project's detailed test suite and our past audits, we believe that such issues are mitigated but cannot be precluded.

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: -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:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	DyDx flash loan callbacks do not check the initiator	**RESOLVED**

The callback function for a DyDx flash loan does not check who initiated the flash loan request. As a result, any attacker can enter _flashLoanLogic with any parameters they choose, possibly significantly disrupting any strategy that uses flash loans.

The code (in FlashLoanHelper) currently reads:

```
/// @dev DyDx calls this function after doing flash loan
function callFunction(
  address,
  /* _sender */
  Account.Info memory, /* _account */
  bytes memory _callData
) external {
  (bytes memory _data, uint256 _repayAmount) = abi.decode(_callData, (bytes,
uint256));
  require(msg.sender == SOLO, "NOT_SOLO");
  _flashLoanLogic(_data, _repayAmount);
}
```

This correctly checks that the caller of this function is SOLO. But it does not check who the caller of SOLO (i.e., the flash loan initiator) was. The first argument of `callFunction` needs to be checked.

Flash loans initiated by untrusted parties have been the attack vector for vulnerabilities in the many millions. In the current Vesper code the use of flash loans is limited, but the above hole has to be patched or it can be disastrous in the future.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Condition in `PoolRewardsUpgrader::_checkResults` is always true	**RESOLVED**

The condition `beforeRewardPerToken[0] == afterRewardPerToken[0]`, which is part of the following require statement, is always true.

```
require(
    beforeRewardToken.length == afterRewardToken.length &&
        beforeRewardToken[0] == afterRewardToken[0] &&
        beforeRewardPerToken[0] == afterRewardPerToken[0],
    "methods-test-failed"
);
```

The reason is that `afterRewardPerToken` is assigned the decoded value of `_beforeResults[2]` instead of `_afterResults[2]` at statement

```
(, address[] memory afterRewardPerToken) = abi.decode(_beforeResults[2],
    (address[], address[]));
```

This is probably a copy-paste error.

LOW SEVERITY:

ID	Description	STATUS
L1	Ineffective restriction on `PoolRewards::updateReward` caller	**RESOLVED**

The `PoolRewards::updateReward` method can only be called by the pool. It is not clear what is the motivation for this restriction. However, method

`PoolRewards::claimReward`, which subsumes the `updateReward` logic, does not set any restriction on the caller (nor can it, since anyone can claim their rewards), suggesting that the former restriction is also unnecessary.		
L2	Optimistic Compound redeem operations might fail	**DISMISSED**
`CompoundXYStrategy::_redeemX` does not adjust the redeemed amount in case it is greater than the balance that the strategy maintains in collateral. This can lead to unnecessary failure when `_redeemX` is called optimistically.		
L3	Aave flash loans do not perform a check for Aave balances	**RESOLVED**
Although DyDx flash loans do check the DyDx liquidity and adjust the request to that amount, Aave flash loans do not do the same. This should not be an issue, just a cause for revert, but for uniformity reasons the logic should best be similar.		
L4	Changing reward durations in staking code can create opportunities for financial benefit	**OPEN**
[This is more of a usage warning, but with monetary implications.] The staking code used in PoolRewards (Synthetix-style staking) gives rewards from past staking in the next period. Such code is typically used with a constant "reward duration" for each reward cycle (although rewards can arrive at any time). The PoolRewards code makes `rewardDuration` a parameter in `_notifyRewardAmount`. Thus, the caller of this function should be careful. If the reward duration changes a lot (e.g., becomes 10x smaller) there is incentive to start staking briefly, just to reap the temporarily higher rewards. This behavior is inherent in the code, so the warning is mostly for the caller. In current code, the only rewards arrive through `Earn::_forwardEarning` and have a duration equal to `dripPeriod`, which can be changed by the governor of the contract.		

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	Duplicate functionality in overridden method	**RESOLVED**
`EarnYearnStrategy::_approveToken` can call `YearnStrategy::approveToken` instead of duplicating its functionality.		
A2	`PaymentStream` state variables can be made immutable	**RESOLVED**
Public state variables `payer` and `token` of the PaymentStream contract can be immutable, as in the current version of the code they can only be set during contract construction.		
A3	`PaymentStreamFactory` storage variable `allStreams` seems unnecessary and perhaps dangerous	**INFO**
There does not seem to be a good reason for keeping storage array `allStreams` in `PaymentStreamFactory`. The address is already an id of a stream, there doesn't seem to be a reason to have an index in an array to identify. Only the length of the array is currently useful. It seems that this field exists for easy off-chain iteration purposes over all streams. If so, it should be noted that untrusted callers of `createStream` can flood the array with an arbitrary number of garbage streams, so the data read should be validated anyway.		
A4	Unnecessary array indexing check	**RESOLVED**
The indexing check in `PaymentStreamFactory::getStream` is unnecessary, since the Solidity compiler should enforce it on the subsequent access.		

```
function getStream(uint256 _idx) external view override returns (address) {
    require(_idx < allStreams.length, "index-exceeds-list-length");
```

```
        return allStreams[_idx];
    }
```

A5	"Magic" constants in the code	**RESOLVED**

The code contains several constants interspersed that should be best concentrated to the beginning of a contract as immutable fields and given easily-understood names. This will improve maintainability, especially in cases where the constants are not set in the constructor. Examples include:

- The constant `1` (to mean SushiSwap) in `_setupOracles()` functions of ConvexSBTCStrategyWBTC, ConvexStrategy.
- The constant `0x4Ddc2D193948926D02f9B1fE9e1daa0718270ED5` for cEth in `CompoundXYStrategy::_getBorrowToken`.

A6	EarnRariFuseStrategyETH does not automatically ensure that it uses ETH	**INFO**

In contrast to other specialized strategies, EarnRariFuseStrategyETH does not seem to enforce its "ETH" part. The code assumes that the cToken used is cETH (e.g., that it receives ETH upon a redeem), but it does not check this. It is not clear if the code is executable in a wrong setup and this issue is probably low-impact, but there is certainly a discrepancy compared to other similar strategies (e.g., EarnYearnStrategyETH).

A7	Floating version pragma in Payment Stream contracts	**RESOLVED**

The floating version `pragma solidity ^0.8.3` is used allowing the contracts to be compiled with any version from 0.8.3 until 0.9.0 of the Solidity compiler. Although the differences between these versions are small, floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment.

A8	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 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.