Vesper Protocol audit

Smart Contract Security Assessment

Dec 14, 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 in the repository https://github.com/bloqpriv/vesper-pools-v3 . The scope of the audit was dual:

1. Recent changes (on multiple files), up to commit 18e2e41b4ba9e493f7e52e887816b10df72745e9 of the develop branch, continuing our previous audit, which had considered changes up to commit 71edb604135c444b7fd46fc0adf558be0b860067 .
2. A re-audit of the main functionality (especially contracts/pool, contracts/strategies/Strategy.sol, example specific strategies).

We have previously audited the Vesper v2 and v3 pools and several strategy contracts. The delta (part 1) of the current audit focuses mostly on verifying newly introduced changes to existing pools and strategies. Specifically:

- The external protocol deposit fee introduction.
- The newly added BuyBack functionality.
- The Curve and Convex base strategies redesign.
- The Curve and Convex 2, 3, 4 and meta pool strategies.
- The New earn pool strategies.
- The Avalanche network Aave strategy.
- Improvements/changes in Alpha, Maker and Vesper strategies.

The total auditing effort was 4 auditor-weeks, split slightly in favor of the re-audit. The auditors reviewing recent changes also 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 protocol has a robust and flexible architecture. Pools and strategies are derived by elaborating general interaction patterns. Accounting is careful, so that, e.g., front-running or price manipulation attacks are preempted. For instance, the accounting of gains and losses, which can affect the price of buying and selling pool shares, is forced to be in a separate transaction, triggered asynchronously by an authorized agent, than the actual pricing of shares at a buy/sell action.

The main threat of omission with this architecture is due to the multiple layers and obligation of overriding functions for proper functioning. There are several points of specialization that every strategy has to define and it is easy to miss some. In an earlier audit, we had recommended clear documentation on the obligations when defining a new strategy or other specialized contract. This recommendation remains.

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:

[No high severity issues]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Crv4PoolStrategy reward token oracle not set up	**OPEN**
	Crv4PoolStrategy receives rewards in CRV and an additional reward token. The strategy can claim this reward token and convert it to another token specified as parameter to function `claimRewardsAndConvertTo`. However, function `_setupOracles` is not overridden in Crv4PoolStrategy to set up the oracle that has to be consulted when swapping from reward token to WETH in function `claimRewardsAndConvertTo`, which can lead to failure if the oracle has not been created beforehand.	
M2	`Crv2PoolStrategy::_depositToCurve` computes incorrect min expected LP amount for `add_liquidity` call	**OPEN**
	When depositing liquidity to Curve, Curve strategies compute the `expectedOut` and `minLpAmount` amounts and provide the maximum of the two as the second argument of the `add_liquidity` call. However, this is not true for the Crv2PoolStrategy that uses the **minimum** of the two amounts as can be seen in the code snippet below.	

```
function _depositToCurve(uint256 _amt) internal virtual override returns (bool) {
  if (_amt != 0) {
    uint256[2] memory _depositAmounts;
    _depositAmounts[collIdx] = _amt;
    uint256 expectedOut =
      _calcAmtOutAfterSlippage(
        IStableSwap2x(address(crvPool)).calc_token_amount(_depositAmounts, true),
        crvSlippage
      );
      uint256 minLpAmount = ((_amt * _getSafeUsdRate()) /
        crvPool.get_virtual_price()) * 10**(18 - coinDecimals[collIdx]);
    if (expectedOut > minLpAmount) expectedOut = minLpAmount;
    // Dedaub: at this point expectedOut = min(expectedOut, minLpAmount)

    // solhint-disable-next-line no-empty-blocks
    try IStableSwap2x(address(crvPool)).
      add_liquidity(_depositAmounts,expectedOut) {}
    catch Error(string memory reason) {
      emit DepositFailed(reason);
      return false;
    }
  ...
}
```

M3	Incorrect error handling in VBoolBase::_withdrawCollateral	OPEN

The following error handling in `VBoolBase::_withdrawCollateral` is incorrect:

```
for (uint256 i; i < _len; i++) {
  ...
  if (_amountNeeded > _debt) {
    // Should not withdraw more than current debt of strategy.
    _amountNeeded = _debt;
  }
  ...
  try IStrategy(_strategy).withdraw(_amountNeeded) {} catch {
    continue;
  }
  ...
  _amountNeeded = _amount - _totalAmountWithdrawn;
}
```

The problem is that `continue` will skip the update of `_amountNeeded`, causing it to have incorrect value in the next iteration. So, if the pool has two strategies, and `withdraw` fails on the first one, `_amountNeeded` might be too small in the second iteration, preventing a withdrawal from the second strategy.

A simple fix would be to move the update of `_amountNeeded` to the beginning of the loop (and make it a local variable).

LOW SEVERITY:

ID	Description	STATUS
L1	False optimization in PoolAccountant	**OPEN**

Several code sites in PoolAccountant contain patterns such as the ones below:

```
function updateExternalDepositFee(...) ... {
  // Read storage once to save gas
  StrategyConfig memory _strategyConfig = strategy[_strategy];
  require(_strategyConfig.active, Errors.STRATEGY_IS_NOT_ACTIVE);
  ...
  emit UpdatedExternalDepositFee(..., _strategyConfig.externalDepositFee, ...);
  ... // Dedaub: no other use of _strategyConfig
}

function updateDebtRatio(...) ... {
  // Read storage once to save gas
  StrategyConfig memory _strategyConfig = strategy[_strategy];
  require(_strategyConfig.active, Errors.STRATEGY_IS_NOT_ACTIVE);
  ...
  totalDebtRatio = (totalDebtRatio - _strategyConfig.debtRatio) + _debtRatio;
  ... // Dedaub: no other use of _strategyConfig
}
```

This is false economy. The StrategyConfig structure is 9 words long. Reading all into memory is much more expensive than reading the 2 or 3 needed individual words from storage. If this pattern was truly introduced for reasons of gas savings, as the comments indicate, it should be reconsidered.

L2	`CrvA3PoolStrategy::_depositToCurve` has unusual form	**OPEN**

[We elevate this issue to "low" to ensure it does not get overlooked, due to its potential financial impact. But we cannot currently find a scenario under which this can be a threat.]

The code of most `_depositToCurve` functions has a 3-step logical form (simplified from real code, below):

```
function _depositToCurve(uint256 amt) internal virtual returns (bool) {
  ...
  uint256 expectedOut =
    _calcAmtOutAfterSlippage(crvPool.calc_token_amount(amt), crvSlippage);
  uint256 minLpAmount = ((amt * _getSafeUsdRate()) / crvPool.get_virtual_price());
  if (expectedOut > minLpAmount) minLpAmount = expectedOut;
  crvPool.add_liquidity(depositAmounts, minLpAmount)
  ...
}
```

That is, first Curve is consulted to get an estimate of the amount-out. Then a second estimate is computed using the Vesper oracle and Curve's `get_virtual_price`, which computes the price of a stableswap LP token relative to an average of its underlying stablecoins. (The `get_virtual_price` function is not subject to manipulation, since it only computes relative to on-chain quantities. The real-world mapping of the price is under the assumption that the tokens are indeed well-pegged stablecoins.) The minimum of the two estimates is then used as a lower bound of the accepted liquidity for the Curve deposit.

However, the `CrvA3PoolStrategy::_depositToCurve` code deviates from the 3-step logic, by omitting the first estimate. (Full code shown.)

```
function _depositToCurve(uint256 amt) internal override returns (bool) {
  if (amt != 0) {
    uint256[3] memory depositAmounts;
    depositAmounts[collIdx] = amt;
    uint256 minLpAmount =
       ((amt * _getSafeUsdRate()) / crvPool.get_virtual_price()) *
       10**(18 - coinDecimals[collIdx]);
    // solhint-disable-next-line no-empty-blocks
    try IStableSwap3xUnderlying(address(crvPool)).
      add_liquidity(depositAmounts, minLpAmount, true) {}
    catch Error(string memory reason) {
      emit DepositFailed(reason);
      return false;
    }
  }
  return true;
}
```

It is not clear to us under which conditions the second estimate alone might not be a sufficiently safe lower bound. For a reliable Vesper oracle, we believe it is safe, in which case the alternative form of the function is perfectly fine. (Note: we do not have access to the current Vesper oracle code, which has evolved since we last audited it.)

However, if there are conditions under which the Vesper+`get_virtual_price` estimate can be financially manipulated, the above discrepancy can be a serious threat.

L3	Incorrect use of `try/catch`	**OPEN**

The following pattern appears in a few places in the Aave strategy (eg. `AaveCore.sol:26`), to store the return value of a function that might revert:

```
try some.func() {
  var = some.func();
} catch() { ... }
```

There are two issues with this code:

- The line `var = some.func()` is not protected by `try`, and in principle `some.func()` might revert the second time it is called, even if the first call succeeds! (Although this is probably impossible in the specific instances of this pattern.)
- It wastes gas by calling `some.func()` twice.

Solidity supports the following form for storing the return value:

```
try some.func() returns (someType ret) {
  var = ret;
} catch() { ... }
```

L4	Asymmetry in the strategy allocation between deposits and withdrawals.	**OPEN**

The ways funds are allocated to strategies is not the same for deposits and withdrawals.

- Deposited tokens are deposited to all strategies based on their `debtRatio`.
- Withdrawn tokens are withdrawn from the first strategy (wrt `_withdrawQueue`).

This might allow an adversary to move funds between strategies by depositing and immediately withdrawing tokens. For instance, consider a hypothetical pool with 10 strategies. The first strategy in `_withdrawQueue` is Aave, which happens to contain a large number of tokens. An adversary wishes to cause a mass liquidation from Aave for some reason (e.g. to tilt interest rates). He can achieve this as follows:

- He deposits a large amount of tokens. These tokens are spit in all 10 strategies.
- Then he asks for a full withdrawal. Since the first strategy is Aave, and it contains enough tokens to cover the user, all tokens are withdrawn from Aave.

In the end no funds are lost, but the adversary has managed to move funds between strategies, at relatively small cost (only the withdrawal fees).

This is not necessarily an attack on Vesper, but it might be useful to keep in mind that such a possibility exists.

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	Emitting event that also performs storage update	**OPEN**

The code in `PoolAccountant::_recalculatePoolExternalDepositFee` performs storage updating in the course of emitting an event. We understand that this saves a temporary variable but it results (at least) in bad readability of the code.

```
function _recalculatePoolExternalDepositFee() internal {
  ...
  emit UpdatedPoolExternalDepositFee(externalDepositFee,
      externalDepositFee = _externalDepositFee);
}
```

ID	Description	STATUS
A2	Code is well covered by external library	**OPEN**

Function `ConvexStrategyBase::_calculateCVXRewards` repeats a calculation from the Convex code and requires references to that code for support.

```
function _calculateCVXRewards(uint256 _claimableCrvRewards) internal view returns
(uint256 _total) {
```

```
// CVX Rewards are minted based on CRV rewards claimed upon withdraw
// This will calculate the CVX amount based on CRV rewards accrued
// without having to claim CRV rewards first
// ref 1:
// https://github.com/convex-eth/platform/blob/main/contracts/contracts/Cvx.sol#L61-L76
// ref 2:
//https://github.com/convex-eth/platform/blob/main/contracts/contracts/Booster.sol#L458-L466

  uint256 _reductionPerCliff = IConvexToken(CVX).reductionPerCliff();
  uint256 _totalSupply = IConvexToken(CVX).totalSupply();
  uint256 _maxSupply = IConvexToken(CVX).maxSupply();
  uint256 _cliff = _totalSupply / _reductionPerCliff;
  uint256 _totalCliffs = 1000;
  ...
```

We see in the Convex code base a library, CvxMining, with a function ConvertCrvToCvx that performs this exact calculation. Reusing such code might be a lot better, e.g., because `totalCliffs` is not immutable in the Convex code, or because the Convex code may change in the future without Vesper noticing. Disclaimer: we are not experts on Convex, developers have to also do due diligence before adopting this suggestion.

A3	Invalid comments explaining the purpose and use of strategies	OPEN

Strategies Convex2PoolStrategy and CrvA3PoolStrategy have wrong “title” comments, due to copy-paste. These comments are also externally visible (plus hinder the understandability of the code), so they should be fixed.

```
/// @title This strategy will deposit collateral token in Curve 3Pool and earn
interest.
abstract contract CrvA3PoolStrategy

/// @title This strategy will deposit collateral token in Curve 4MetaPool and
stake lp token to convex.
abstract contract Convex2PoolStrategy
```

A4	Unnecessary temporary memory arrays	OPEN

In the _init functions of CrvPoolStrategyBase, CrvsBTCPoolStrategy, Crv4PoolStrategy, CrvA3PoolStrategy, the temporary arrays _coins and _coinDecimals are unnecessary: values can be written directly to the storage arrays, as done in other strategies. The cost is not great, however.

```
function _init(address _crvPool, uint256 _n) internal virtual {
  address[] memory _coins = new address[](_n);
  uint256[] memory _coinDecimals = new uint256[](_n);
  for (uint256 i = 0; i < _n; i++) {
    _coins[i] = IStableSwapUnderlying(_crvPool).coins(i);
    _coinDecimals[i] = IERC20Metadata(_coins[i]).decimals();
  }
  coins = _coins;
  coinDecimals = _coinDecimals;
}
```

A5	No check for number of coins in Curve pool	OPEN

In the above _init code or other initialization code (in CrvPoolStrategyBase or others), there does not seem to exist a check to ensure that the _crvPool is one with _n coins. Admittedly, this is a setup issue, which can be enforced externally, but analogous checks can be found in the code in different circumstances.

```
  for (uint256 i = 0; i < _n; i++) {
    _coins[i] = IStableSwapUnderlying(_crvPool).coins(i);
    _coinDecimals[i] = IERC20Metadata(_coins[i]).decimals();
  }
```

A6	Unnecessary approve call	OPEN

In Crv4PoolStrategy and Crv4MetaPoolStrategy, function _approveToken, the first of the approvals below (to 0) seems unnecessary. It does not prevent the ERC20 approval attack, and is superseded by the very next call.

```
function _approveToken(uint256 _amount) internal virtual override {
  ...
  for (uint256 j = 0; j < swapManager.N_DEX(); j++) {
    IERC20(_rewardToken).safeApprove(address(swapManager.ROUTERS(j)), 0);
    IERC20(_rewardToken).safeApprove(address(swapManager.ROUTERS(j)), _amount);
  }
}
```

A7	Unnecessary actions and over-complex logic in infinite approval	**OPEN**

The combination of infinite approval logic in BuyBack+UsingSwapManager is strange and performs unnecessary work.

```
// BuyBack
function doInfinityApproval(IERC20 _unwrapped) external onlyKeeper {
  _doInfinityApprovalIfNeeded(_unwrapped, type(uint256).max);
}

// UsingSwapManager
function _doInfinityApprovalIfNeeded(IERC20 _asset, uint256 _amountToSwap)
internal {
  for (uint256 i = 0; i < swapManager.N_DEX(); i++) {
    uint256 _allowance = IERC20(_asset).allowance(address(this),
                                  address(swapManager.ROUTERS(i)));
    if (_allowance < _amountToSwap) {
      IERC20(_asset).safeIncreaseAllowance(address(swapManager.ROUTERS(i)),
                           type(uint256).max - _allowance);
  }
}
```

First, the point of infinite approval is to not have to re-do it. If the approved amount is always checked to be max, the approval is done again after each transferFrom, negating any benefits of infinite approval.

Second, there is no need to use `safeIncreaseAllowance` when the total allowance will be infinite. A plain `safeApprove` would suffice (and not do unnecessary extra work, since `safeIncreaseAllowance` just re-adds what the above code just subtracted: the current allowance).

Finally, logic-wise it is strange to have infinite allowance but only if the current one is not enough. Indeed, the one client of `_doInfinityApprovalIfNeeded` does not provide a precise `_amountToSwap`, but infinity. Unless there may be other uses of this function in the future, the whole functionality above is best replaced with a straightforward infinite `safeApprove`.

A8	Code duplication	**OPEN**

A few strategies override methods `_init` and/or `_setupOracles` of base strategies without actually extending or changing the base implementation, leading to unnecessary code duplication. We list the strategies and the methods below:

- CrvsBTCPoolStrategy: `_init` and `_setupOracles` are implemented in `CrvPoolStrategyBase`
- Crv4PoolStrategy: `_init` is implemented in `CrvPoolStrategyBase`
- ConvexSBTCStrategyWBTC: `_setupOracles` is implemented in `ConvexStrategy` and `CrvPoolStrategyBase`

A9	The use of `AddressListFactory` is costly	

The use of `AddressListFactory`, although certainly elegant, is costly in gas. Each list is a separate contract, hence initializing a VPool requires creating several new contracts. Moreover, modifiers such as `onlyKeeper`, used extensively throughout the code, perform external calls to the list contract. Implementing lists within VPool would likely lead to gas savings.

A10	Unlimited approvals from the Strategies to the Pool	**OPEN**

Strategies set an unlimited approval to the Pool contract. This is not a problem when the Pool's code is safe. But it weakens the separation between contracts, and could make it easier to exploit a potential vulnerability in case a bug is found or introduced in the future. It would be preferable to only set approvals when needed and for the

specific amount. Moreover, one should have this in mind when re-assessing the security of VPool in the future.		
A11	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 is: ● Memory layout corruption can happen when using abi.decode for the deserialization of two-dimensional arrays.		

DISCLAIMER

ABOUT DEDAUB