Vector Reserve Security Audit

Report Version 0.1

February 1, 2024

Conducted by:
George Hunter, Independent Security Researcher

Table of Contents

1 About George Hunter **3**

2 Disclaimer **3**

3 Risk classification **3**
3.1 Impact . . . 3
3.2 Likelihood . . . 3
3.3 Actions required by severity level . . . 3

4 Executive summary **4**

5 Findings **5**
5.1 Low . . . 5
5.1.1 Vector token transfers may unexpectedly revert . . . 5
5.1.2 Restaked LSTs with callbacks could break balances accounting in vETH . . . 6
5.1.3 Improper tokens allowance management . . . 6
5.1.4 sVEC index can be left uninitialized . . . 7
5.1.5 Admin will withdraw funds that should remain in the contract balance . . . 8

1 About George Hunter

George Hunter is a proficient and reputable independent smart contract security researcher with over 50 solo and team security engagements contributing to the security of numerous smart contract protocols in the past 2 years. Previously held roles include Lead Smart Contract Auditor at Paladin Blockchain Security and Smart Contract Engineer at Nexo. He has audited smart contracts for clients such as LayerZero, Euler, TraderJoe, Maverick, Ambire, and other leading protocols.

2 Disclaimer

Audits are a time-, resource-, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

George Hunter was engaged by Vector Reserve to review their smart contract protocol during the period from January 25, 2024, to January 31, 2024.

Overview

Project Name	Vector Reserve
Repository	https://github.com/vectorreserve/vr
Commit hash	b758e436d417effb3ded9dcdf9344de6c3b27523
Resolution	-
Methods	Manual review

Timeline

-	January 25, 2024	Audit kick-off
v0.1	February 1, 2024	Preliminary report

Scope

contracts/tokens/StakedVectorETH.sol
contracts/tokens/Vector.sol
contracts/tokens/VectorETH.sol
contracts/tokens/sVEC.sol
contracts/staking/VECStaking.sol
contracts/reward/RewardDistributor.sol
contracts/treasury/VectorTreasury.sol
contracts/bonds/VectorBonding.sol
contracts/bonds/BondDepo.sol

Issues Found

High risk	0
Medium risk	0
Low risk	5

5 Findings

5.1 Low

5.1.1 Vector token transfers may unexpectedly revert

Severity: *Low*

Context: Vector.sol#L511

Description: The Vector contract represents a fee-on-transfer token that swaps accumulated fees to ETH and distributes them among different parties. On every transfer, the following check is made:

```
if (
    canSwap && swapEnabled && !swapping && !automatedMarketMakerPairs[from] &&
    !_isExcludedFromFees[from] && !_isExcludedFromFees[to]
) {
    swapping = true;
    swapBack();
    swapping = false;
}
```

The `swapBack()` function distributes the accumulated fees from VEC transfers by using the ratio between the fees accumulated by the different parties. One of the fee destination addresses is the UniswapV2 pool of VEC:WETH. The following formula is used to determine the portion of the contract's balance that should go to the pool (only VEC side):

```
uint256 liquidityTokens = (contractBalance * tokensForLiquidity) /
    totalTokensToSwap /
    2;
```

The rest of the accumulated VEC tokens are then swapped to ETH and distributed among the other parties:

```
uint256 amountToSwapForETH = contractBalance - liquidityTokens;
```

There's a possible edge case in which the `liquidityTokens` variable is greater than the `contractBalance` in the above arithmetic operation which causes it to revert due to underflow and block the `swapBack()` method.

If the `contractBalance` is high enough, it is capped at 20x the `swapTokensAtAmount()`:

```
if (contractBalance > swapTokensAtAmount() * 20) {
    contractBalance = swapTokensAtAmount() * 20;
}
```

However, this doesn't modify the accumulated fees meaning that the code assumes they will always be less than the updated `contractBalance` which is not always the case.

Recommendation: Given that the contract is already deployed and immutable, and this edge case is highly unlikely to occur due to the needed preconditions, a fix is not mandatory. If the above scenario ever occurs on chain, the admin can simply disable the swaps and set the fees to 0 in order to unblock the transfers.

Resolution: Acknowledged.

5.1.2 Restaked LSTs with callbacks could break balances accounting in vETH

Severity: *Low*

Context: VectorETH.sol#L89-L94

Description: The VectorETH uses restaked LST tokens as collateral which are whitelisted by the contract's admin.

Upon deposit/mint, the tokens can be staked either in the contract itself, or in an external contract that "manages" this restaked LST token:

```
if (routeRestakedLSTTo[_restakedLST] == address(0)) {
    IERC20(_restakedLST).safeTransferFrom(
        _msgSender(),
        address(this),
        _amount
    );
} else {
    IERC20(_restakedLST).safeTransferFrom(
        _msgSender(),
        routeRestakedLSTTo[_restakedLST],
        _amount
    );
    restakedLSTManaged[_restakedLST] += _amount;
}
```

A potential attack vector is that the `restakedLSTManaged` mapping accounting can be manipulated if a restaked LST token with callback on transfer is added (i.e. ERC777 token). If such token is added, an attacker would be able to reenter from the `safeTransferFrom` call on line 89 into the `redeem` method, and basically stake no tokens, but still increase the `restakedLSTManaged` mapping. The exploit could be repeated as many times as needed to cause Denial-of-Service to functions like the `updateDeposit` and `manageRestakedLST`.

Recommendation: Carefully validate the implementation of any LST tokens added to the VectorETH contract and document the above behavior. Alternatively, consider implementing re-entrancy guards.

Resolution: Acknowledged.

5.1.3 Improper tokens allowance management

Severity: *Low*

Context: sVEC.sol#L237-L238, Vector.sol#L366-L368

Description: The Vector token contract implements a `burnFrom` functionality that allows an account that a user approved to transfer tokens on their behalf also burn this amount of tokens:

```
function _burnFrom(address account, uint256 amount) internal {
    uint256 decreasedAllowance_ = allowance(account, msg.sender) - amount;

    _approve(account, msg.sender, decreasedAllowance_);
    _burn(account, amount);
}
```

An inconsistency is present in this function as it reduces the allowance even if it is set to type(uint256).max which is not done in `transferFrom`:

```
function _spendAllowance(address owner, address spender, uint256 amount) internal
    virtual {
    uint256 currentAllowance = allowance(owner, spender);
    if (currentAllowance != type(uint256).max) {
        require(currentAllowance >= amount, "ERC20: insufficient allowance");
        unchecked {
            _approve(owner, spender, currentAllowance - amount);
        }
    }
}
```

Recommendation: Consider removing lines 237 and 238 of the `sVEC` contract as well as implementing a max allowance check in `Vector._burnFrom`.

Resolution: Acknowledged.

5.1.4 sVEC index can be left uninitialized

Severity: *Low*

Context: sVEC.sol#L83-L86, sVEC.sol#L106

Description: The `sVEC` implements both a constructor and an `initialize` method as well a `setIndex` method which sets the `INDEX` storage variable.

The problem is that the `initialize` method uses the `initializer` address (which is set in the constructor) for replay protection in the following way:

```
constructor() ERC20("Staked Vector", "sVEC") {
    initializer = msg.sender;
    // ...
}

function initialize(address _stakingContract) external {
    require(
        msg.sender == initializer,
        "Initializer:  caller is not initializer"
    );

    // ...

    initializer = address(0);
}
```

The `setIndex` function implements the same access control check meaning that it will not be accessible if the `initialize` method is called first causing the `INDEX` variable to be left uninitialized.

Recommendation: Consider checking that the initial `INDEX` has been set before executing the `initialize` function logic.

Resolution: Acknowledged.

5.1.5 Admin will withdraw funds that should remain in the contract balance

Severity: *Low*

Context: Vector.sol#L695-L709, RewardDistributor.sol#L123-L132, VectorETH.sol#L227-L238

Description: The Vector, VectorETH and RewardDistributor contracts implement a set of two functions that allow the admin of the contracts to withdraw any stuck native or ERC20 tokens. All 3 contract are expected hold and manage tokens that should not be withdrawable by the contract admin. However, the whole balance of the contract is withdrawn in these functions. Also no input validation is present in the Vector and RewardDistributor contracts and only a simple check for whether the token is a restaked LST token is present in VectorETH which is not sufficient.

Recommendation: Consider passing the `amount` the admin intends to withdraw as well as checking that they do not try to withdraw tokens accounted in the contract.

Resolution: Acknowledged.