BlockSec

Security Audit Report for Alpaca Delta Neutral Vault

Date: Feb 18, 2022

Version: 1.0

Contact: contact@blocksecteam.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.02.18 21:00:29 +08'00'

Contents

Report Manifest

Item	Description
Client	Alpaca
Target	Alpaca Delta Neutral Vault

Version History

Version	Date	Description
1.0	Feb 18, 2022	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values of the repo [1] during the audit are shown in the following.

Contract Name	Stage	Commit SHA
delta-neutral-vault	Initial	cb13e32fe5a4ba6f63b0235bd4624715592e4abe
delta-neutral-vault	Final	e7c3899416e86a045011febec7a5cc986176e406

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report do not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team).

[1] https://github.com/alpaca-finance/bsc-alpaca-contract/tree/feat/delta-neutral-vault/contracts/8.10

We also manually analyze possible attack scenarios with independent auditors to cross-check the result.

- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find `seven` potential issues in the smart contract. We also have `three` recommendations, as follows:

- High Risk: 1
- Medium Risk: 2
- Low Risk: 4
- Recommendations: 3

ID	Severity	Description	Category	Status
1	Medium	Potential Precision Loss	Software Security	Fixed
2	Low	Unreturned Values	Software Security	Fixed
3	Low	Unchecked Initialization Parameters	Software Security	Fixed
4	High	Unlimited Withdraw Value	DeFi Security	Fixed
5	Low	Potential Locking of Native Tokens	DeFi Security	Fixed
6	Medium	Unchecked Price	DeFi Security	Fixed
7	Low	Potential Locked Tokens	DeFi Security	Acknowledged
8	-	Avoiding Duplicated Calculations	Recommendation	Fixed
9	-	Avoiding Inconsistency Checks in the Worker Contracts	Recommendation	Fixed
10	-	Considering the Impact of Transaction Ordering Dependency	Recommendation	Acknowledged

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential Precision Loss

Status Fixed

Description In the `DeltaNeutralVault` contract and the two worker contracts, there are cases of multiplying after division which may cause precision losses. For example,

1. In the `withdraw()` function of the `DeltaNeutralVault` contract, the variable `_withdrawValue` is divided by `1e18`. After that, this variable is passed into the `_withdrawHealthCheck()` function and used as a multiplier. If the result of `_stableWithdrawValue` + `_assetWithdrawValue` fall into the range of `1e18` - `10e18`, the precision loss might be up to almost 50%, which can lead to the failure of invoking the `_withdrawHealthCheck()` function.

```
        uint256 _withdrawValue;
        {
        uint256 _stableWithdrawValue = _stableTokenBack * priceHelper.getTokenPrice(
            stableToken);
        uint256 _assetWithdrawValue = _assetTokenBack * priceHelper.getTokenPrice(
            assetToken);
        _withdrawValue = (_stableWithdrawValue + _assetWithdrawValue) / 1e18;
```

```
            }

            // sanity check
            _withdrawHealthCheck(_withdrawValue, _positionInfoBefore, _positionInfoAfter);
```

Listing 2.1: withdraw():DeltaNeutralVault.sol

```
            uint256 _stableExpectedWithdrawValue = (_withdrawValue *_positionInfoBefore.
                stablePositionEquity)/_totalEquityBefore;
            uint256 _stableActualWithdrawValue = _positionInfoBefore.stablePositionEquity -
                _positionInfoAfter.stablePositionEquity;

            if(!Math.almostEqual(_stableActualWithdrawValue, _stableExpectedWithdrawValue,
                _toleranceBps)){
              revert UnsafePositionValue();
            }
            uint256 _assetExpectedWithdrawValue = (_withdrawValue *_positionInfoBefore.
                assetPositionEquity)/_totalEquityBefore;
```

Listing 2.2: _withdrawHealthCheck():DeltaNeutralVault.sol

2. In the `_withdrawHealthCheck()` function of the `DeltaNeutralVault` contract, the two variables, i.e., `_stableExpectedWithdrawValue` and `_assetExpectedWithdrawValue`, are calculated by being first divided by `_totalEquityBefore`, and then multiplied in the `Math.almostEqual()` function. The precision loss brought here may affect the result of the `_withdrawHealthCheck()` function as well.

Impact It may cause (severe) precision losses.

Suggestion Apply the proper method to perform the calculation.

2.1.2 Unreturned Values

Status Fixed

Description The return value of the `claim()` function is not properly asigned.

```
  /// @notice Claim Alpaca reward of stable vault and asset vault
  function claim() external returns (uint256, uint256) {
    uint256 rewardStableVault = _claim(IVault(stableVault).fairLaunchPoolId());
    uint256 rewardAssetVault = _claim(IVault(assetVault).fairLaunchPoolId());
  }
```

Listing 2.3: DeltaNeutralVault.sol

Impact The return values of the `claim()` function is always 0.

Suggestion Return the variables properly.

2.1.3 Unchecked Initialization Parameters

Status Fixed

Description There are no checks on the parameters of the `initialize()` function of the `DeltaNeutralVault` contract.

```
function initialize(
    string calldata _name,
    string calldata _symbol,
    address _stableVault,
    address _assetVault,
    address _stableVaultWorker,
    address _assetVaultWorker,
    address _lpToken,
    address _alpacaToken,
    IPriceHelper _priceHelper,
    IDeltaNeutralVaultConfig _config
  ) external initializer {
    OwnableUpgradeable.__Ownable_init();
    ReentrancyGuardUpgradeable.__ReentrancyGuard_init();
    ERC20Upgradeable.__ERC20_init(_name, _symbol);

    stableVault = _stableVault;
    assetVault = _assetVault;

    stableToken = IVault(_stableVault).token();
    assetToken = IVault(_assetVault).token();
    alpacaToken = _alpacaToken;

    stableVaultWorker = _stableVaultWorker;
    assetVaultWorker = _assetVaultWorker;

    lpToken = _lpToken;

    priceHelper = _priceHelper;
    config = _config;
  }
```

Listing 2.4: DeltaNeutralVault.sol

Impact N/A

Suggestion Check the initialization parameters in the `initialize()` function.

2.2 DeFi Security

2.2.1 Unlimited Withdraw Value

Status Fixed

Description

The logic of the `withdraw()` function in `DeltaNeutralVault` is implemented as follows:

1. Burning specified number of shares from the caller.
2. Partially or entirely closing the position by calling the `_execute()` function for parameter `_data`.
3. Calculating the real value withdrawn, and returning the corresponding tokens to the user.

However, in this function, there is no check between the user shares burnt and the actual value withdrawn by invoking `_execute()`.

```
/// @notice Withdraw from delta neutral vault.
/// @param _shareAmount Amount of share to withdraw from vault.
/// @param _minStableTokenAmount Minimum stable token shareOwner expect to receive.
/// @param _minAssetTokenAmount Minimum asset token shareOwner expect to receive.
/// @param _data The calldata to pass along to the proxy action for more working context.
function withdraw(
  uint256 _shareAmount,
  uint256 _minStableTokenAmount,
  uint256 _minAssetTokenAmount,
  bytes calldata _data
) public onlyEOAorWhitelisted nonReentrant returns (uint256 _withdrawValue) {

  address _shareOwner = msg.sender;
  PositionInfo memory _positionInfoBefore = positionInfo();
  Outstanding memory _outstandingBefore = _outstanding();

  uint256 _shareValue = shareToValue(_shareAmount);
  _burn(_shareOwner, _shareAmount);

  {
    (uint8[] memory actions, uint256[] memory values, bytes[] memory _datas) = abi.decode(
      _data,
      (uint8[], uint256[], bytes[])
    );
    _execute(actions, values, _datas);
  }

  PositionInfo memory _positionInfoAfter = positionInfo();
  Outstanding memory _outstandingAfter = _outstanding();

  // transfer funds back to shareOwner
  uint256 _stableTokenBack = stableToken == config.getWrappedNativeAddr()
    ? _outstandingAfter.nativeAmount - _outstandingBefore.nativeAmount
    : _outstandingAfter.stableAmount - _outstandingBefore.stableAmount;
  uint256 _assetTokenBack = assetToken == config.getWrappedNativeAddr()
    ? _outstandingAfter.nativeAmount - _outstandingBefore.nativeAmount
    : _outstandingAfter.assetAmount - _outstandingBefore.assetAmount;

  if (_stableTokenBack < _minStableTokenAmount) {
    revert InsufficientTokenReceived(stableToken, _minStableTokenAmount, _stableTokenBack);
  }
  if (_assetTokenBack < _minAssetTokenAmount) {
    revert InsufficientTokenReceived(assetToken, _minAssetTokenAmount, _assetTokenBack);
  }

  _transferTokenToShareOwner(_shareOwner, stableToken, _stableTokenBack);
  _transferTokenToShareOwner(_shareOwner, assetToken, _assetTokenBack);

  uint256 _withdrawValue;
  {
    uint256 _stableWithdrawValue = _stableTokenBack * priceHelper.getTokenPrice(stableToken);
    uint256 _assetWithdrawValue = _assetTokenBack * priceHelper.getTokenPrice(assetToken);
```

```
        _withdrawValue = (_stableWithdrawValue + _assetWithdrawValue) / 1e18;
      }

      // sanity check
      _withdrawHealthCheck(_withdrawValue, _positionInfoBefore, _positionInfoAfter);
      _outstandingCheck(_outstandingBefore, _outstandingAfter);

      emit LogWithdraw(_shareOwner, _stableTokenBack, _assetTokenBack);
      return _withdrawValue;
    }
```

Listing 2.5: DeltaNeutralVault.sol

Impact Malicious users can withdraw more value than the shares they owned.

Suggestion Check the actual withdraw value with the burnt shares.

2.2.2 Potential Locking of Native Tokens

Status Fixed

Description In the `deposit()` function of the `DeltaNeutralVault` contract, the `_transferTokenToVault()` function is called to transfer both native tokens and ERC-20 tokens to the vault. However, if either `assetToken` or `stableToken` is `config.getWrappedNativeAddr()` (i.e., `WBNB` for the Binance Smart Chain), this function will directly deposit the transferred value to `WBNB`. However, the function does not check whether `msg.value` is the same as the `_amount` passed as the parameter. As such, if `msg.value` is larger than `_amount`, it may cause some native tokens (i.e., `BNB`) being locked in this contract.

```
    /// @notice Get token from msg.sender.
    /// @param _token token to transfer.
    /// @param _amount amount to transfer.
    function _transferTokenToVault(address _token, uint256 _amount) internal {
      if (_token == config.getWrappedNativeAddr()) {
        IWETH(config.getWrappedNativeAddr()).deposit{ value: _amount }();
      } else {
        SafeToken.safeTransferFrom(_token, msg.sender, address(this), _amount);
      }
    }
```

Listing 2.6: DeltaNeutralVault.sol

Impact The native tokens will be locked in the contract if the `msg.value` is larger than `_amount`.

Suggestion Add sanity checks to prevent the locking.

2.2.3 Unchecked Price

Status Fixed

Description The `DeltaNeutralVault` contract relies on a `PriceHelper` contract to request price information from Chainlink to calculate the prices for the tokens provided by users and LP tokens. Specifically, the obtained price is associated with a timestamp. However, this timestamp is not verified in the `PriceHelper` contract. As a result, the obtained price may be outdated and hence invalid.

```
function getTokenPrice(address tokenAddress) public view returns (uint256) {
  (uint256 price, uint256 lastTimestamp) = chainLinkPriceOracle.getPrice(tokenAddress, usd);
  return price;
}
```

Listing 2.7: PriceHelper.sol

Impact The prices returned by the `PriceHelper` contract may be invalid.

Suggestion Check returned timestamp in the `PriceHelper` contract.

2.2.4 Potential Locked Tokens

Status Acknowledged

Description

To provide flexibility to the `DeltaNeutralVault` contract, the actual operations are wrapped as a raw calldata parameter to the functions. As a result, a user needs to specify the amount of tokens to transfer to this contract. Therefore, there is a possibility that the executed operations use less token than the actual amount deposited by the user. As there is no way of withdrawing these extraneous tokens, they will be locked in the contract.

Impact Some of the tokens provided by the users may be locked in the `DeltaNeutralVault` contract.

Suggestion N/A

Feedback from the Developers We can reinvest the left over native token by wrapping and depositing WBNB in positions. Our users will benefit from the reinvest since the equity will increase but the total share remain the same. Since the contract is upgradable if there are some funds left in the contract, we can upgrade the contract then extract it.

2.3 Additional Recommendation

2.3.1 Avoiding Duplicated Calculations

Status Fixed

Description In the `positionInfo()` function of the `DeltaNeutralVault` contract, the `_positionDebtValue()` function is invoked multiple times with the same parameters. As the `_positionDebtValue()` function has several external calls, the duplicated calls may lead to unnecessary gas consumption.

```
  function positionInfo() public view returns (PositionInfo memory) {
    return
      PositionInfo({
        stablePositionEquity: _positionEquity(stableVault, stableVaultWorker, stableVaultPosId),
        stablePositionDebtValue: _positionDebtValue(stableVault, stableVaultPosId),
        assetPositionEquity: _positionEquity(assetVault, assetVaultWorker, assetVaultPosId),
        assetPositionDebtValue: _positionDebtValue(assetVault, assetVaultPosId)
      });
  }
```

Listing 2.8: DeltaNeutralVault.sol

```
function _positionEquity(address _vault, address _worker, uint256 _posId) internal view returns
    (uint256) {
  uint256 _positionValue = _positionValue(_worker);
  uint256 _positionDebtValue = _positionDebtValue(_vault, _posId);
  if( _positionValue < _positionDebtValue){
    return 0;
  }
  return _positionValue - _positionDebtValue;
}
```

Listing 2.9: DeltaNeutralVault.sol

Impact The duplicated calculations may cause extraneous gas usage.

Suggestion Remove the duplicated calculations.

2.3.2 Avoiding Inconsistency Checks in the Worker Contracts

Status Fixed

Description In the `work()` function of the `DeltaNeutralPancakeWorker02` contract, there is a guard in the `_reinvest()` call to check whether `treasuryAccount` and `treasuryBountyBps` are set. However, there is no corresponding check in the `DeltaNeutralMdexWorker02` contract.

```
  function work(
      uint256 id,
      address user,
      uint256 debt,
      bytes calldata data
    ) external override onlyWhitelistedCaller(user) onlyOperator nonReentrant {
      // 1. If a treasury configs are not ready. Not reinvest.
      if (treasuryAccount != address(0) && treasuryBountyBps != 0)
        _reinvest(treasuryAccount, treasuryBountyBps, actualBaseTokenBalance(), reinvestThreshold
            );
```

Listing 2.10: DeltaNeutralPancakeWorker02.sol

```
  function work(
      uint256 id,
      address user,
      uint256 debt,
      bytes calldata data
    ) external override onlyWhitelistedCaller(user) onlyOperator nonReentrant {
      // 1. reinvest
      _reinvest(treasuryAccount, treasuryBountyBps, actualBaseTokenBalance(), reinvestThreshold);
```

Listing 2.11: DeltaNeutralMdexWorker02.sol

Impact N/A

Suggestion Remove the unnecessary check.

2.3.3 Considering the Impact of Transaction Ordering Dependency

Status Acknowledged

Description In functions deposit() and withdraw() of the DeltaNeutralVault contract, the parameter _data are crucial in managing the position of DeltaNeutralVault in Vault. We assume that it may be pre-calculated by the frontend for users. However, the calculation will be based on the state which may be affected by the order of the transactions inside one block. In this case, it may cause the failure of the transactions.

Impact N/A

Feedback from the Developers Yes, the failure due to transaction ordering is expected behavior. For example, if the prices from oracle deviate from DEXes too much, it will affect the equity value of the positions and transaction should revert.