code4rena

Tornado

Smart Contract Security Assessment

Version 2.0

Audit dates: Jun 03 — Jun 05, 2024

Audited by: MiloTruck
HollaDieWaldFee100

Contents

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Tornado

Tornado Blast v1 is a sophisticated TG bot designed for Blast, facilitating easy and super-fast trading and interaction within the Blast ecosystem. We recognize the difficulty of navigating numerous dexes and finding opportunities amidst a deluge of information. Our goal is to simplify your navigation through this storm of opportunities. Therefore, we propose to offer this all-encompassing tool, enabling effortless trading and interaction with protocols.

2.2 Scope

Repository	kairos-loan/token-contracts-for-audit
Commit Hash	e39491d605cb17d8f93fdf04de946d1be47418b6
Mitigation Hash	704dbc2d5f5828ee454aec9ee9d7dd10c1b14263

2.3 Audit Timeline

DATE	EVENT
Jun 03, 2024	Audit start
Jun 05, 2024	Audit end
Oct 29, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	3
Informational	3
Total Issues	9

3. Findings Summary

ID	DESCRIPTION	STATUS
M-1	Fee and profit payments into `RevenueSharingVault` can be sandwiched	Resolved

M-2	Initial mint amount of TRNDO is wrong	Resolved
M-3	\`VestingMaster\`: Banning account causes future rewards in \`RevenueSharingVault\` to get lost and tokens are stuck	Resolved
L-1	\`IndividualVestingVault.claimableTokenAmount()\` can underflow due to rounding in ERC4626 Vault	Resolved
L-2	Maximum buy/sell tax is not checked	Resolved
L-3	\`VestingMaster\`: Check that \`schedule.cliffEndDate\` is greater than \`block.timestamp\`	Resolved
I-1	\`RevenueSharingVault\` is vulnerable to share inflation attacks	Acknowledged
I-2	\`IndividualVestingVault.unstakableTokenAmount()\` can be simplified	Resolved
I-3	Remove unnecessary \`pool\` equals \`owner()\` condition in \`TornadoBlastBotToken.taxIfTaxIsActive()\`	Resolved

4. Findings

4.1 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] Fee and profit payments into `RevenueSharingVault` can be sandwiched

Severity: Medium	Status: Resolved

Context: [RevenueSharingVault.sol#L12-L24]

Description: TRNDO transfer fees and trading profits are sent to `RevenueSharingVault` where the shares of stakers appreciate. Rewards to stakers are paid out immediately. As a result, if large payouts are made, it can be profitable to make a flash-deposit to earn rewards without having staked TRNDO for any considerable amount of time.

Such behavior acts as a net transfer of rewards from unsophisticated to sophisticated stakers. Consider that the buy and sell fee payments to the Vault can be sandwiched within a single transaction by constructing a transaction consisting of:

1. deposit into the Vault
2. buy or sell TRNDO and incur the buy / sell fee
3. withdraw from the Vault

Payments of trading gains are harder to sandwich because they can only be triggered by an external transaction, and Blast has no mempool that allows front-running. Still, the same considerations apply that users that have staked for a short period of time can earn an outsized portion of rewards.

Recommendation: It is recommended to send rewards to a separate `VaultDistributor` contract that ensures rewards are paid out slowly. Available rewards are snapshotted and paid out linearly over a configurable timeframe. At the end of the timeframe, available rewards are snapshotted again.

This approach requires that additional functions in ERC4626 must be overridden.

Changes in `RevenueSharingVault`:

```
--- a/apps/contracts/src/tornadoToken/RevenueSharingVault.sol
+++ b/apps/contracts/src/tornadoToken/RevenueSharingVault.sol
@@ -7,12 +7,18 @@ import { ERC4626 } from
"@openzeppelin/contracts/token/ERC20/extensions/ERC4626.
 import { TornadoBlastBotToken } from "./TornadoBlastBotToken.sol";
 import { BlastGasAndYield } from "../commons/BlastGasAndYield.sol";

+import {VaultDistributor} from "../VaultDistributor.sol";
+
 /// @dev send tornado blast tokens to this contract to redistribute them
to stakers
 /// @dev treasury MUST stake a significant amount first to avoid future
share/tokenAmount slippage
 contract RevenueSharingVault is ERC4626, BlastGasAndYield {
+    VaultDistributor vaultDistributor;
     constructor(
-        TornadoBlastBotToken tornadoBlastToken
-    ) ERC4626(tornadoBlastToken) ERC20("Staked Tornado Blast Token",
"stTRNDO") {}
+        TornadoBlastBotToken tornadoBlastToken,
+        VaultDistributor _vaultDistributor
+    ) ERC4626(tornadoBlastToken) ERC20("Staked Tornado Blast Token",
"stTRNDO") {
+        vaultDistributor = _vaultDistributor;
+    }

     function _update(address from, address to, uint256 value) internal
override {
         // allow mint and burn, disallow transfers
@@ -21,4 +27,32 @@ contract RevenueSharingVault is ERC4626,
BlastGasAndYield {
         }
         super._update(from, to, value);
     }
+
+    function setVaultDistributor(VaultDistributor _vaultDistributor)
external onlyOwner {
+        vaultDistributor = _vaultDistributor;
+    }
+
+    function totalAssets() public view override returns (uint256) {
+        return _asset.balanceOf(address(this)) +
vaultDistributor.pendingRewards();
+    }
+
+    function deposit(uint256 assets, address receiver) public override
returns (uint256) {
+        vaultDistributor.processRewards();
+        super.deposit(assets, receiver);
+    }
+
+    function mint(uint256 shares, address receiver) public override
returns (uint256) {
+        vaultDistributor.processRewards();
+        super.mint(shares, receiver);
+    }
+
+    function withdraw(uint256 assets, address receiver, address owner)
public override returns (uint256) {
+        vaultDistributor.processRewards();
+        super.withdarw(assets, receiver, owner);
+    }
+
+    function redeem(uint256 shares, address receiver, address owner)
public override returns (uint256) {
+        vaultDistributor.processRewards();
+        super.redeem(shares, receiver, owner);
+    }
 }
```

New `VaultDistributor` contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.25;

import { Ownable } from "@openzeppelin/contracts/access/Ownable.sol";
import { VestingUtils } from "./Vesting/VestingUtils.sol";
import { ERC4626 } from
"@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";


contract VaultDistributor is Ownable, VestingUtils {
    uint256 payoutPeriod;
    uint256 activePayoutPeriod;
    uint256 lastTimestamp;
    uint256 allPaidOutTimestamp;
    uint256 rewardBalance;

    constructor(ERC4626 _tokenizedVault) VestingUtils(_tokenizedVault)
Ownable(msg.sender) {
        payoutPeriod = 1 weeks;
        lastTimestamp = block.timestamp;
        activePayoutPeriod = payoutPeriod;
        allPaidOutTimestamp = block.timestamp;
    }

    function setPayoutPeriod(uint256 newPayoutPeriod) external onlyOwner
{
        require(payoutPeriod > 0, "payoutPeriod cannot be zero");
        payoutPeriod = newPayoutPeriod;
    }


    function processRewards() external {
        uint256 amountToPay = pendingRewards();
        if (amountToPay > 0) {
            vestedToken.transfer(address(tokenizedVault), amountToPay);
        }

        if (block.timestamp >= allPaidOutTimestamp) {
            activePayoutPeriod = payoutPeriod;
            allPaidOutTimestamp = block.timestamp + activePayoutPeriod;
            rewardBalance = vestedToken.balanceOf(address(this));
        }
        lastTimestamp = block.timestamp;
    }

    function pendingRewards() public view returns(uint256) {
        uint256 timestampNow = block.timestamp;
        if (block.timestamp > allPaidOutTimestamp) {
            timestampNow = allPaidOutTimestamp;
        }

        uint256 timePassed = timestampNow - lastTimestamp;
        uint256 amountToPay = rewardBalance * timePassed /
activePayoutPeriod;
        return amountToPay;
    }

}
```

Tornado: All fixes implemented by this commit.

Zenith: Verified. The implementation has been simplified and the payout period is hardcoded. In addition, `LinearDistributor` inherits from `BlastGasAndYield` such that Blast gas can be claimed.

[M-2] Initial mint amount of TRNDO is wrong

Severity: Medium	Status: Resolved

Context: TornadoBlastBotToken.sol#L40

Description:

In the constructor of `TornadoBlastBotToken`, 100 million tokens should be minted to the owner:

```
_mint(msg.sender, 10e8 * 10 ** decimals()); // 100 million
```

However, the code specifies `10e8`, which is one billion and not 100 million.

Recommendation:

Consider using `100_000_000` instead of `10e8`, which is much more readeable:

```
- _mint(msg.sender, 10e8 * 10 ** decimals()); // 100 million
+ _mint(msg.sender, 100_000_000 * 10 ** decimals()); // 100 million
```

Tornado: Fixed.

Zenith: Verified.

[M-3] `VestingMaster`: Banning account causes future rewards in `RevenueSharingVault` to get lost and tokens are stuck

Severity: Medium	Status: Resolved

Context:

- VestingMaster.sol#L39-L42
- IndividualVestingVault.sol#L45-L51

Description: It is possible for the `owner` of `VestingMaster` to ban accounts from calling `IndividualVestingVault.claim()`. As a result of this, the shares that belong to the banned account, are never withdrawn and just remain owned by `IndividualVestingVault`. This means that any revenue earned on these shares is lost forever.

Recommendation: It is recommended to redeem the Vault shares for TRNDO and send them back to `VestingMaster` when an account is banned. Thereby, no TRNDO token get stuck and revenue is not lost.

```
--- a/apps/contracts/src/Vesting/IndividualVestingVault.sol
+++ b/apps/contracts/src/Vesting/IndividualVestingVault.sol
@@ -42,6 +42,10 @@ contract IndividualVestingVault is VestingUtils,
Initializable {
         tokenizedVault.withdraw(amountToClaim, msg.sender,
address(this));
     }

+    function banAccount() external onlyVestingMaster() {
+        tokenizedVault.redeem(tokenizedVault.balanceOf(address(this)),
msg.sender, address(this));
+    }
+
     function claimableTokenAmount() public view returns (uint256) {
         if (vaultIsBanned()) {
             return 0;

--- a/apps/contracts/src/Vesting/VestingMaster.sol
+++ b/apps/contracts/src/Vesting/VestingMaster.sol
@@ -36,9 +36,14 @@ contract VestingMaster is IVestingMaster,
VestingUtils, Ownable, MinimalProxyFac
         }
     }
```

```
+    function withdrawTornadoTokens(address recipient) external onlyOwner
{
+        vestedToken.transfer(recipient,
vestedToken.balanceOf(address(this)));
+    }
+
     function banAccount(address account) external onlyOwner {
         require(!claimingIsProhibitedFor[account]);
         claimingIsProhibitedFor[account] = true;
+        vaultOf[account].banAccount();
     }
```

Tornado: Fixed.

Zenith: Verified. Instead of redeeming the shares to `VestingMaster`, they are redeemed to the `owner` of `VestingMaster`.

4.2 Low Risk

A total of 3 low risk findings were identified.

[L-1] `IndividualVestingVault.claimableTokenAmount()` can underflow due to rounding in ERC4626 Vault

Severity: Low	Status: Resolved

Context: IndividualVestingVault.sol#L45-L51

Description: When withdrawing assets, `RevenueSharingVault` rounds up the amount of shares to burn. As a result, if a user withdraws amount x of assets from the vault, their value of assets in the vault can drop by more than x.

The finding can be confirmed by adding the following test to `Vesting.t.sol`:

```
function testClaimableTokenAmountUnderflow() public {
        initBobVault();
        skip(2 days);
        grantGainedYieldToSharingVault(929384722818138423489);
        claimAs(bob);
        // @audit here the calculation underflows due to rounding up the
shares to burn in the first calculation
        claimAs(bob);
    }
```

The severity of the issue is "Low" since the underflow is only temporary and as more funds are vested, the calculation can be executed successfully again.

Recommendation: Check for the underflow condition and return zero instead.

```
         uint256 lockedAmount = tokenAmountInitiallyStaked() -
claimableNowFromLinearVesting();
-        return unstakableTokenAmount() - lockedAmount; // includes extra
yield on top of linear vested tokens
+        uint256 unstakableTokenAmount = unstakableTokenAmount();
+        if (unstakableTokenAmount < lockedAmount) {
+            return 0;
+        } else {
+            return unstakableTokenAmount() - lockedAmount; // includes
extra yield on top of linear vested tokens
+        }
```

```
        }
```

Tornado: Fixed.

Zenith: Verified

[L-2] Maximum buy/sell tax is not checked

Severity: Low Status: Resolved

Context: TornadoBlastBotToken.sol#L46-L52

Description:

When the `TornadoBlastBotToken` contract is first deployed, the maximum buy and sell tax is set to 5% in `MAX_SELL_TAX_SHARE` and `MAX_BUY_TAX_SHARE`. However, `setSellTaxShare()` and `setBuyTaxShare()` do not check these values:

```
function setSellTaxShare(Ray newTaxShare) external onlyOwner {
    sellTaxShare = newTaxShare;
}

function setBuyTaxShare(Ray newTaxShare) external onlyOwner {
    buyTaxShare = newTaxShare;
}
```

This allows the owner to set the buy/sell tax to any arbitrary percentage.

Recommendation:

Ensure `newTaxShare` is not larger than `MAX_SELL_TAX_SHARE`/`MAX_BUY_TAX_SHARE` in their respective functions:

```
  function setSellTaxShare(Ray newTaxShare) external onlyOwner {
+     require(MAX_SELL_TAX_SHARE.gte(newTaxShare));
      sellTaxShare = newTaxShare;
  }
```

```
  function setBuyTaxShare(Ray newTaxShare) external onlyOwner {
+     require(MAX_BUY_TAX_SHARE.gte(newTaxShare));
      buyTaxShare = newTaxShare;
  }
```

Tornado: Fixed.

Zenith: Verified

[L-3] `VestingMaster`: Check that `schedule.cliffEndDate` is greater than `block.timestamp`

Severity: Low	Status: Resolved

Context: VestingMaster.sol#L72-L76

Description: The checks in `VestingMaster.validateSchedule()` fail to validate that `schedule.cliffEndDate` is greater than `block.timestamp`. A schedule with `schedule.cliffEndDate < block.timestamp` would immediately make some of the locked funds claimable. According to the client, this is not intended and it is recommended to add the missing sanity check.

Recommendation:

```diff
     function validateSchedule(Schedule memory schedule) internal view {
         require(schedule.totalClaimableTokens > 0);
-        require(schedule.linearVestingEndDate > block.timestamp);
+        require(schedule.cliffEndDate > block.timestamp);
         require(schedule.linearVestingEndDate > schedule.cliffEndDate);
     }
```

Tornado: Fixed.

Zenith: Verified

4.3 Informational

A total of 3 informational findings were identified.

[I-1] `RevenueSharingVault` is vulnerable to share inflation attacks

Severity: Informational	Status: Acknowledged

Context: RevenueSharingVault.sol#L10-L24

Description:

`RevenueSharingVault` inherits Openzeppelin's `ERC4626` without overriding `_decimalsOffset()`:

```
/// @dev send tornado blast tokens to this contract to redistribute them
to stakers
/// @dev treasury MUST stake a significant amount first to avoid future
share/tokenAmount slippage
contract RevenueSharingVault is ERC4626, BlastGasAndYield {
    constructor(
        TornadoBlastBotToken tornadoBlastToken
    ) ERC4626(tornadoBlastToken) ERC20("Staked Tornado Blast Token",
"stTRNDO") {}

    function _update(address from, address to, uint256 value) internal
override {
        // allow mint and burn, disallow transfers
        if (from != address(0) && to != address(0)) {
            revert("staked tornado blast tokens are not transferrable");
        }
        super._update(from, to, value);
    }
}
```

This means that the vault's virtual assets and shares are both `1` (ie. the vault starts with 1 asset and 1 share). If the treasury does not deposit into the vault first, an attacker can donate TRNDO to the vault to cause future deposits to lose funds, for example:

- Assume `RevenueSharingVault` is newly deployed.
- Attacker deposits 1 TRNDO into the vault, receiving 1 share in return.
- Attacker transfers `100e18` TRNDO into the vault.

- Owner calls `VestingMaster.vestTokensForNewAccounts()` to vest `10e18` TRNDO for 20 individual addresses:
  - `10e18` TRNDO is deposited into the vault 20 times.
  - The amount of shares minted for each deposit is `10e18 * (1 + 1) / (100e18 + 1) = 0`.
  - All 20 deposits receive no shares for the vested TRNDO.

In the scenario above, the TRNDO deposited will accrue to the attacker's 1 share, allowing him to make a profit.

Recommendation: Ensure that the treasury stakes some amount of TRNDO into the vault before tokens are distributed to users.

Tornado: Acknowledged.

[I-2] `IndividualVestingVault.unstakableTokenAmount()` can be simplified

Severity: Informational Status: Resolved

Context: [IndividualVestingVault.sol#L83-L86]

Description:

`IndividualVestingVault.unstakableTokenAmount()` calculates the maximum amount of assets that can be withdrawn from `tokenizedVault` as such:

```
    function unstakableTokenAmount() internal view returns (uint256) {
        VaultShares stakedAmount =
VaultShares.wrap(tokenizedVault.balanceOf(address(this)));
        return
tokenizedVault.convertToAssets(VaultShares.unwrap(stakedAmount));
    }
```

This can be simplified using `ERC4626.maxWithdraw()`.

Recommendation:

Use `maxWithdraw()` instead, which returns the maximum amount of assets that can be withdrawn:

```
  function unstakableTokenAmount() internal view returns (uint256) {
-     VaultShares stakedAmount =
VaultShares.wrap(tokenizedVault.balanceOf(address(this)));
-     return
tokenizedVault.convertToAssets(VaultShares.unwrap(stakedAmount));
+     return tokenizedVault.maxWithdraw(address(this));
  }
```

This is essentially the same logic as the current implementation, just shorter.

Tornado: Fixed.

Zenith: Verified.

[I-3] Remove unnecessary `pool` equals `owner()` condition in `TornadoBlastBotToken.taxIfTaxIsActive()`

Severity: Informational	Status: Resolved

Context: [TornadoBlastBotToken.sol#L99]()

Description: `owner()` will never be the same as an active pool, so the condition is redundant.

Recommendation:

```
     ) internal returns (uint256 remainingAmount) {
-        if (pool == owner() || taxIsDisabled(taxShare) ||
isBuyBack(transfer)) {
+        if (taxIsDisabled(taxShare) || isBuyBack(transfer)) {
             return transfer.value;
         }
         return performTax(transfer, taxShare);
```

Tornado: [Fixed]().

Zenith: Verified.