Aburra Security Review

Pashov Audit Group

Conducted by: ast3ros, pontifex, 0xbepresent, peanuts

May 20th 2024 - May 23th 2024

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Introduction 3
4. About Aburra 3
5. Risk Classification 4
   5.1. Impact 4
   5.2. Likelihood 4
   5.3. Action required for severity levels 5
6. Security Assessment Summary 5
7. Executive Summary 6
8. Findings 9
   8.1. Critical Findings 9
      [C-01] Malicious token creator can block the token buy/sells 9
   8.2. Medium Findings 12
      [M-01] Users can evade paying fees 12
      [M-02] Token depreciation due to ETH donating 12
      [M-03] No Slippage Protection in buyWithReferral 13
      [M-04] Potential loss of LP Fees 14
      [M-05] Unintended transfer of creator fees to zero address 17
      [M-06] Potential asset misdirection due to race condition in token transfer 19
      [M-07] tradeTokenMinSupply must not be zero 21
   8.3. Low Findings 23
      [L-01] Lack of ethVirtualSupply param check 23
      [L-02] Hardcoded referral fee 23
      [L-03] Use Ownable2Step rather than Ownable 24
      [L-04] ETH can be transferred to the Purple DAO address even if it was not set 24
      [L-05] Deadline doesn’t protect against long-pending transactions 25

[L-06] moveToAMM can be reverted or lead to fund losses 26
[L-07] The total fee can add up to above 10000 27
[L-08] x * y = k invariant not maintained 28
[L-09] Malicious creators can gas-grief users 29

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Project-Aegis-xyz/tokenFactory** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Aburra

The TokenFactory smart contract allows users to create coins that trade on a bonding curve (xy=k with virtual ETH liquidity), with the option to migrate the token to a Uniswap v3 pool if the price reaches a certain threshold within a specified period; otherwise, the liquidity is donated to the Purple DAO's treasury.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - c16e8f6355d80415de4ae75a86e6a88e33fe4038

fixes review commit hash - b9d932c33a58adafcd2edb7b89d362d60dcdbf04

Scope

The following smart contracts were in scope of the audit:

- `FixedSupplyERC20`
- `TokenFactory`
- `IWETH9`

7. Executive Summary

Over the course of the security review, ast3ros, pontifex, 0xbepresent, peanuts engaged with Aburra to review Aburra. In this period of time a total of **17** issues were uncovered.

Protocol Summary

Protocol Name	Aburra
Repository	https://github.com/Project-Aegis-xyz/tokenFactory
Date	May 20th 2024 - May 23th 2024
Protocol Type	Bonding Curve token sale

Findings Count

Severity	**Amount**
Critical	1
Medium	7
Low	9
Total Findings	**17**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Malicious token creator can block the token buy/sells	Critical	Resolved
[M-01]	Users can evade paying fees	Medium	Acknowledged
[M-02]	Token depreciation due to ETH donating	Medium	Acknowledged
[M-03]	No Slippage Protection in buyWithReferral	Medium	Resolved
[M-04]	Potential loss of LP Fees	Medium	Resolved
[M-05]	Unintended transfer of creator fees to zero address	Medium	Resolved
[M-06]	Potential asset misdirection due to race condition in token transfer	Medium	Resolved
[M-07]	tradeTokenMinSupply must not be zero	Medium	Resolved
[L-01]	Lack of ethVirtualSupply param check	Low	Resolved
[L-02]	Hardcoded referral fee	Low	Resolved
[L-03]	Use Ownable2Step rather than Ownable	Low	Resolved
[L-04]	ETH can be transferred to the Purple DAO address even if it was not set	Low	Resolved
[L-05]	Deadline doesn’t protect against long-pending transactions	Low	Resolved
[L-06]	moveToAMM can be reverted or lead to fund losses	Low	Acknowledged

[L-07]	The total fee can add up to above 10000	Low	Resolved
[L-08]	x * y = k invariant not maintained	Low	Resolved
[L-09]	Malicious creators can gas-grief users	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Malicious token creator can block the token `buy/sells`

Severity

Impact: High

Likelihood: High

Description

When a token is created, the creator can specify the `creatorFeeRecipient`, which will be used for sending the corresponding fees. Then in the `TokenFactory` contract, the functions `buyWithReferral`, `buyWithReferralWithExactEth`, and `sellWithReferral` use the `_sendFees` function to handle the transfer of fees:

```
File: TokenFactory.sol
490:     function sellWithReferral(
491:         address erc20Address,
492:         uint256 tokenAmountToSell,
493:         uint256 minEthReceivedAfterFees,
494:         address referral
495:     ) external nonReentrant validToken(erc20Address) {
...
536:         // send fees
537:         _sendFees
  (erc20Address, protocolFee, creatorFee, referralFee, referral);
...
```

The `_sendFees` function includes a `revert` statement, located at lines `TokenFactory#L806-809`, that could be triggered by a token creator to intentionally fail the transaction.

```
File: TokenFactory.sol
806: (
  boolcreatorFeeSuccess,

) = creatorFeeRecipients[erc20Address].call{value: creatorFee, gas: 100_000}(""
807: if (!creatorFeeSuccess) {
808:     revert TokenFactory_CreatorFeeTransferFailed();
809: }
```

This issue allows the token creator to control who can buy or sell tokens by strategically causing the transaction to fail, the token creator can leverage this to prevent token sales, ensuring that the token he created can be pumped to create the `Uniswap pool`. The following test demonstrates that the malicious creator can cause the `buyWithReferral` call to be reverted:

```
// File: TokenFactoryForkTest_BuyAndSell.t.sol
    function test_buyMaliciousCreatorDOS() public {
        string memory name = "TestToken";
        string memory symbol = "TT";
        uint256 initialBuyerBalance = 10 ether;
        // Malicious creator contract
        MaliciousCreator customCreatorFeeRecipient = new MaliciousCreator();
        //
        // 1. Token creation using the `malicious creator` contract
        vm.startPrank(buyer);
        address newTokenAddress = factory.initializeToken(name, symbol, address
          (customCreatorFeeRecipient));
        vm.deal(buyer, initialBuyerBalance);
        //
        // 2. Malicious creator can block purchases
        vm.expectRevert(bytes4(keccak256("TokenFactory_CreatorFeeTransferFailed
          ()")));
        factory.buyWithReferral{value: 10 ether}
          (newTokenAddress, 1e7 ether, address(0));
        vm.stopPrank();
    }
```

The `MaliciousCreator` contract:

```
contract MaliciousCreator {
    receive() external payable {
        // custom revert
        revert();
    }
}
```

Recommendations

OPTION 1:

The `TokenFactory::_sendFees` function should be modified so that it does not revert when the `creatorFee` transfer fails:

```
(
      boolcreatorFeeSuccess,

    ) = creatorFeeRecipients[erc20Address].call{value: creatorFee, gas: 100_000}(""
    if (!creatorFeeSuccess) {
        emit CreatorFeeTransferFailed(erc20Address, creatorFee);
    }
```

OPTION 2:

Consider having a mapping for each creator and the tokenAddress, so that if such an attack happens, the protocol can single out the malicious creator and set his creatorFee to zero, without impacting the other creators.

Since there is already a mapping of creatorFeeRecipients[ERC20Token] to the creator, recommend changing the mapping to point to a struct, with the creator and the creator fees.

8.2. Medium Findings

[M-01] Users can evade paying fees

Severity

Impact: Medium

Likelihood: Medium

Description

The `TokenFactory` charges fees for both buying and selling tokens.

```
// initialize fees
        protocolFeeBips = 150;
        creatorFeeBips = 250;

        protocolFeeWithReferralBips = 100;
        creatorFeeWithReferralBips = 250;
        referralFeeBips = 50;
```

The issue here is that users can evade paying both fees by simply trading their tokens in the open market (e.g. Dex, Uniswap pairs, etc...)

Recommendations

Consider pausing all transfers when `curve.isPaused` is `false`.

[M-02] Token depreciation due to ETH donating

Severity

Impact: High

Likelihood: Low

Description

The `TokenFactory.pauseAndWithdrawToPurple` function pauses a token and donates its ETH to Purple DAO when not reach the price target within a certain period of time. So all token holders lose the possibility to sell tokens back. It means decreasing the token price to zero, and losing all investments.

```
/// @param erc20Address The address of a valid bonding curve token.
    function pauseAndWithdrawToPurple
      (address erc20Address) external onlyOwner nonReentrant validToken(erc20Address)
        BondingCurveData storage curve = curves[erc20Address];
>>      curve.isPaused = true;
        uint256 ethBalance = curve.ethBalance;
        if (ethBalance > 0) {
>>          curve.ethBalance = 0;
            (
              boolethSent,

            ) = purpleAddress.call{value: ethBalance, gas: 100_000}(""
            if (!ethSent) revert TokenFactory_EthTransferFailed();
            emit EthWithdrawnToPurple(erc20Address, ethBalance);
        }
    }
```

Recommendations

Consider using this function only for rescuing excess ETH from the token curve if ETH was refunded by `UNISWAP_V3_POSITION_MANAGER` and implementing a separate function for curve pausing when a certain time passed.

[M-03] No Slippage Protection in buyWithReferral

Severity

Impact: High

Likelihood: Low

Description

When users buy tokens using the `buyWithReferral` function, they specify the token amount to buy and send the corresponding amount of ETH. There isn't slippage protection due to the assumption that the maximum price users will pay is `tokenAmountToBuy/msg.value`, allowing users to control the price they pay.

However, users can still end up paying a higher price than expected (slippage) because:

- The number of tokens available for purchase might be less than the `tokenAmountToBuy` if the remaining token supply is less than the `tokenAmountToBuy`.
- As the available token supply decreases, the price of the token increases.

In a scenario where a user intends to buy 10e18 tokens with 1 ETH (price is 0.1 ETH per 1e18 tokens), but another user front-runs and buys 9e18 tokens before them, the initial user will only receive 1e18 tokens and end up paying more than 0.1 ETH, which is more than the intended price of 0.1 ETH per 1e18 tokens.

```
function buyWithReferral
      (address erc20Address, uint256 tokenAmountToBuy, address referral)
        public
        payable
        nonReentrant
        validToken(erc20Address)
    {
        ...
        uint256 tokenAmountLeft = getTokenAmountLeft(erc20Address);
        if
        //(tokenAmountToBuy >= tokenAmountLeft) { // @audit amount to buy can be less
            tokenAmountToBuy = tokenAmountLeft;
            // pause the curve when all available tokens have been bought
            curves[erc20Address].isPaused = true;
            emit BondingCurveCompleted(erc20Address);
        }
        ...
    }
```

Recommendations

Allow users to specify the minimum amount of tokens they will receive to avoid slippage.

[M-04] Potential loss of LP Fees

Severity

Impact: High

Likelihood: Low

Description

When a token sale completes, the corresponding `curvePool` is set to `paused`. Subsequently, the owner can invoke `TokenFactory::moveToAMM` to create a pool on Uniswap and add liquidity (TokenFactory#L888-905). This action mints a Uniswap position NFT and stores its ID (TokenFactory#L858) for fee collection later. However, Uniswap may return ETH if the full amount isn't used for liquidity, which can incorrectly leave the curve's ETH balance non-zero (`curve.ethBalance`) (TokenFactory#L917-931):

```
File: TokenFactory.sol
861:     function _createPoolAndAddLiquidity(
862:         address erc20Address,
863:         uint256 ethBalance,
864:         uint256 tokenBalance,
865:         uint16 slippageBips
866:     ) internal returns (uint256 tokenId) {
...
...
916:         // Remove ERC20 allowance and update ETH balance (if refunded)
917:         if (amount0Added < amount0) {
918:             if (token0 == WETH9) {
919:                 curve.ethBalance = amount0 - amount0Added;
920:             } else {
921:                 TransferHelper.safeApprove(token0, address
  (UNISWAP_V3_POSITION_MANAGER), 0);
922:             }
923:         }
924:
925:         if (amount1Added < amount1) {
926:             if (token1 == WETH9) {
927:                 curve.ethBalance = amount1 - amount1Added;
928:             } else {
929:                 TransferHelper.safeApprove(token1, address
  (UNISWAP_V3_POSITION_MANAGER), 0);
930:             }
931:         }
932:     }
```

This non-zero balance can erroneously allow `TokenFactory::moveToAMM` to be called again since the function checks if there's enough liquidity to proceed without reverting (TokenFactory#L854-856):

```
File: TokenFactory.sol
840:     function moveToAMM(address erc20Address, uint16 slippageBips)
841:         external
842:         onlyOwner
843:         nonReentrant
844:         validToken(erc20Address)
845:     {
846:         if (!curves[erc20Address].isPaused) {
847:             revert TokenFactory_BondingCurveNotPaused();
848:         }
849:         BondingCurveData storage curve = curves[erc20Address];
850:
851:         uint256 ethBalance = curve.ethBalance;
852:         uint256 tokenBalance = IERC20(erc20Address).balanceOf(address
  (this));
853:
854:         if (ethBalance == 0 || tokenBalance == 0) {
855:             revert TokenFactory_InsufficientLiquidity();
856:         }
857:
858:         tokenIds[erc20Address] = _createPoolAndAddLiquidity
  (erc20Address, ethBalance, tokenBalance, slippageBips);
859:     }
```

Consider the following scenario:

1. The token pool completes, and the owner initially calls `moveToAMM`.
2. A Uniswap pool is created, and a position is minted, but some ETH is refunded, leaving `curve.ethBalance` non-zero `TokenFactory#L919`.

```
File: TokenFactory.sol
861:     function _createPoolAndAddLiquidity(
862:         address erc20Address,
863:         uint256 ethBalance,
864:         uint256 tokenBalance,
865:         uint16 slippageBips
866:     ) internal returns (uint256 tokenId) {
...
...
916:         // Remove ERC20 allowance and update ETH balance (if refunded)
917:         if (amount0Added < amount0) {
918:             if (token0 == WETH9) {
919:                 curve.ethBalance = amount0 - amount0Added;
920:             } else {
921:                 TransferHelper.safeApprove(token0, address
  (UNISWAP_V3_POSITION_MANAGER), 0);
922:             }
923:         }
...
932:     }
```

3. The owner inadvertently calls `moveToAMM` again, perhaps triggered by an automatic script or misunderstanding in the off-chain process. An attacker could exploit this by sending tokens to meet the liquidity check `(TokenFactory#L854-856)`, allowing the creation of another position. This

new position ID overwrites the old one (TokenFactory#L858), losing the ability to claim LP fees from the initially created position in step 1.

Ultimately, the contract will only be able to claim `LP fees` from the position created in `step 3`, as this position is the only one stored in `tokenIds[erc20Address]`. The position created in step 2 will no longer be able to claim its `LP fees`.

Recommendations

To mitigate this issue, it is recommended to handle ETH refunds in a way that prevents them from being reused for another `moveToAMM` call.

[M-05] Unintended transfer of creator fees to zero address

Severity

Impact: Medium

Likelihood: Medium

Description

During the creation of a `tokenPool` a user can specify the `creatorFeeRecipient` (TokenFactory#L266):

```
File: TokenFactory.sol
262:      */
263:     function initializeTokenAndBuy(
264:         string calldata name,
265:         string calldata symbol,
266:         address creatorFeeRecipient,
267:         uint256 initialBuyAmount
268:     ) external payable returns (address newTokenAddress) {
269:         newTokenAddress = initializeToken
  (name, symbol, creatorFeeRecipient);
...
...
```

The issue arises when the token creator specifies the `creatorFeeRecipient` as `address(0)` causing that any accumulated creator fees will be transferred to `address(0)`, effectively burning these fees.

```
File: TokenFactory.sol
794:     function _sendFees(
795:         address erc20Address,
796:         uint256 protocolFee,
797:         uint256 creatorFee,
798:         uint256 referralFee,
799:         address referralAddr
800:     ) internal {
...
806:         (
  boolcreatorFeeSuccess,

) = creatorFeeRecipients[erc20Address].call{value: creatorFee, gas: 100_000}(""
807:         if (!creatorFeeSuccess) {
808:             revert TokenFactory_CreatorFeeTransferFailed();
809:         }
...
818:     }
```

This can be problematic as there may be users who decide not to receive fees (`creatorFeeRecipient=address(0)`); however, the system will still calculate those fees and send them to `address(0)`. The following test demonstrates how a user initializes the token using `address(0)` as the `creatorFeeRecipient`, then a purchase of the token is made, and the fees intended for the creator are sent to `address(0)`:

```
function test_creatorAddressZero() public {
        string memory name = "TestToken";
        string memory symbol = "TT";
        uint256 initialBuyerBalance = 10 ether;
        //
        // 1. Token creation using the `malicious creator` contract
        vm.startPrank(buyer);
        address newTokenAddress = factory.initializeToken(name, symbol, address
          (0));
        vm.deal(buyer, initialBuyerBalance);
        //
        // 2. Buyer buys and the creator fees are sent to zero address
        factory.buyWithReferral{value: 10 ether}
          (newTokenAddress, 1e7 ether, address(0));
    }
```

Recommendations

It is recommended to validate that a token cannot be created with `creatorFeeRecipient=address(0)`. Alternatively, it could be evaluated that if a user decides not to receive fees from the creation of the token, then these `creatorFees` should be sent to the protocol.

[M-06] Potential asset misdirection due to race condition in token transfer

Severity

Impact: High

Likelihood: Low

Description

When a token has sold all of its tokens, the token is paused and marked as "completed" (TokenFactory#L388-393). This allows the assets to be moved to a `Uniswap Pool` using the `TokenFactory::moveToAMM` function:

```
File: TokenFactory.sol
371:     function buyWithReferral
  (address erc20Address, uint256 tokenAmountToBuy, address referral)
372:         public
373:         payable
374:         nonReentrant
375:         validToken(erc20Address)
376:     {
...
...
387:         uint256 tokenAmountLeft = getTokenAmountLeft(erc20Address);
388:         if (tokenAmountToBuy >= tokenAmountLeft) {
389:             tokenAmountToBuy = tokenAmountLeft;
390:             // pause the curve when all available tokens have been bought
391:             curves[erc20Address].isPaused = true;
392:             emit BondingCurveCompleted(erc20Address);
393:         }
```

On the other hand, the function `TokenFactory::pauseAndWithdrawToPurple` assists in pausing the token pool and donating the ETH to PurpleDAO.

```
File: TokenFactory.sol
936:     function pauseAndWithdrawToPurple
  (address erc20Address) external onlyOwner nonReentrant validToken(erc20Address) {
937:         BondingCurveData storage curve = curves[erc20Address];
938:         curve.isPaused = true;
939:         uint256 ethBalance = curve.ethBalance;
940:         if (ethBalance > 0) {
941:             curve.ethBalance = 0;
942:
  (bool ethSent,) = purpleAddress.call{value: ethBalance, gas: 100_000}("");
943:             if (!ethSent) revert TokenFactory_EthTransferFailed();
944:             emit EthWithdrawnToPurple(erc20Address, ethBalance);
945:         }
946:     }
```

The issue arises when the `pauseAndWithdrawToPurple` function is called while there are still tokens available for sale and someone completes the token pool. The following sequence of events can trigger the misdirection of assets:

1. The token still has some tokens for sale.
2. The owner calls the `pauseAndWithdrawToPurple` function but their transaction stays in the mempool.
3. A user buys all the remaining tokens, leaving the token as "completed" and paused, waiting to be moved to a Uniswap pool.
4. The transaction from `step 2` is executed, sending all tokens to the `PurpleDAO`.

This sequence of events is incorrect as the Uniswap pool should have been created and the assets transferred to it, rather than to the PurpleDAO.

The following test demonstrates how a token pool that is `paused` will have its assets sent to PurpleDAO using the `pauseAndWithdrawToPurple` function:

```
function test_WithdrawToPurpleWhenCurveIsCompleted() public {
        //
        // 1. Create token and complete the bonding curve
        address tokenAddress = _initializeToken(buyer, 1e9 ether);
        //
        // 2. The Bonding curve is paused (completed)
        (uint256 initialBondingCurveEthBalance, bool isPaused) = factory.curves
          (tokenAddress);
        assertEq(isPaused, true);
        //
        // 3. Owner calls `pauseAndWithdrawToPurple`
        vm.startPrank(contractOwner);
        uint256 initialPurpleBalance = factory.purpleAddress().balance;
        factory.pauseAndWithdrawToPurple(tokenAddress);
        vm.stopPrank();
        //
        // 4. Balances are transferred to `purple`
        (uint256 finalBondingCurveEthBalance,) = factory.curves(tokenAddress);
        uint256 finalContractBalance = address(factory).balance;
        uint256 finalPurpleBalance = factory.purpleAddress().balance;
        // check balance and eth withdraw
        assertEq(finalContractBalance, 0);
        assertEq(finalBondingCurveEthBalance, 0);
        assertEq(
          finalPurpleBalance,
          initialPurpleBalance+initialBondingCurveEthBalance
        );
    }
```

Additionally, another potential attack vector is that the creation of a Uniswap pool may not be possible due to the `deadline=15 minutes` parameter in `TokenFactory#L902`. This makes it susceptible to the function `pauseAndWithdrawToPurple()` being called by an off-chain automated script

execution, which could divert resources to the `PurpleDAO`. This is incorrect as those resources should be sent to a Uniswap pool.

Recommendation

To mitigate this issue, it is recommended to introduce a validation step in the `pauseAndWithdrawToPurple` function to ensure that the token is not in a state where it is waiting to be moved to a Uniswap pool:

```
function pauseAndWithdrawToPurple
      (address erc20Address) external onlyOwner nonReentrant validToken(erc20Address)
        BondingCurveData storage curve = curves[erc20Address];
++      if (curve.isPaused) revert TokenFactory_BondingCurvePaused();
        curve.isPaused = true;
        uint256 ethBalance = curve.ethBalance;
        if (ethBalance > 0) {
            curve.ethBalance = 0;
            (
              boolethSent,

            ) = purpleAddress.call{value: ethBalance, gas: 100_000}(""
            if (!ethSent) revert TokenFactory_EthTransferFailed();
            emit EthWithdrawnToPurple(erc20Address, ethBalance);
        }
    }
```

[M-07] tradeTokenMinSupply must not be zero

Severity

Impact: High

Likelihood: Low

Description

When creating a new token, the creator needs to set the initial supply and the minimum supply. Take note that the minimum supply cannot be zero, otherwise the AMM calculation will not work (divide by zero) and all the ETH that is spent on buying the tokens cannot be transferred to the uniswap position as `curves[erc20Address]` cannot be paused, making the ETH stuck in the contract.

```
uint256 newX = x - tokenAmount;
        @audit - the newX will be zero, which will not work
>       uint256 newYScaled = (k * _PRECISION_MULTIPLIER) / newX;
```

Recommendations

Check that `tradeTokenMinSupply` is not zero when calling `initializeCustomToken()`.

```
function initializeCustomToken(
        if (tradeTokenInitSupply <= tradeTokenMinSupply) {
            revert TokenFactory_InvalidParams();
        }
       require(tradeTokenMinSupply != 0, "Minimum supply cannot be zero");
```

8.3. Low Findings

[L-01] Lack of `ethVirtualSupply` param check

The `TokenFactory.initializeCustomToken` function does not check if the `ethVirtualSupply` param is zero or too small. This can cause unexpected results in the price calculation and make the token prone to inflation attacks.

```
function initializeCustomToken(
        string calldata name,
        string calldata symbol,
        address creatorFeeRecipient,
        uint256 tradeTokenInitSupply,
        uint256 tradeTokenMinSupply,
>>      uint256 ethVirtualSupply
    ) public nonReentrant returns (address newTokenAddress) {
        if (tradeTokenInitSupply <= tradeTokenMinSupply) {
            revert TokenFactory_InvalidParams();
        }

        IERC20 newToken = new FixedSupplyERC20
          (name, symbol, tradeTokenInitSupply);
        newTokenAddress = address(newToken);

        params[newTokenAddress] = BondingCurveParams({
            tradeTokenInitSupply: tradeTokenInitSupply,
            tradeTokenMinSupply: tradeTokenMinSupply,
>>          ethVirtualSupply: ethVirtualSupply
        });
```

Consider checking if the `ethVirtualSupply` value is in the reasonable range (e.g. greater than 0).

[L-02] Hardcoded referral fee

Though `protocolFeeWithReferralBips` and `creatorFeeWithReferralBips` fees can be changed the `referralFeeBips` is hardcoded.

```
function initialize(
        address initialOwner,
        INonfungiblePositionManager _uniswapV3PositionManager,
        IUniswapV3Factory _uniswapV3Factory,
        address _protocolFeeRecipient,
        address _weth9,
        address _purple
    ) public initializer {
<...>
        protocolFeeWithReferralBips = 100;
        creatorFeeWithReferralBips = 250;
>>      referralFeeBips = 50;
```

Consider setting the difference between `protocolFeeBips` and `protocolFeeWithReferralBips` as a new `referralFeeBips` value in the `setProtocolFeeBips` function.

[L-03] Use Ownable2Step rather than Ownable

`Ownable2Step` and `Ownable2StepUpgradeable` prevent the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner permissions actively accept via a contract call of its own.

```
contract TokenFactory is
    Initializable,
    UUPSUpgradeable,
    IERC721Receiver,
    ERC721HolderUpgradeable,
    ReentrancyGuardUpgradeable,
>>  OwnableUpgradeable
```

[L-04] ETH can be transferred to the Purple DAO address even if it was not set

Though the address of the Purple DAO's treasury should be set with `TokenFactory.initialize` function, the address can be left as `0` and be set later with the `setPurpleAddress` function. The problem is that the `pauseAndWithdrawToPurple` function does not check if the Purple DAO address is not zero and ETH can be lost.

```
function initialize(
        address initialOwner,
        INonfungiblePositionManager _uniswapV3PositionManager,
        IUniswapV3Factory _uniswapV3Factory,
        address _protocolFeeRecipient,
        address _weth9,
        address _purple
    ) public initializer {
        __Ownable_init(initialOwner);
        __UUPSUpgradeable_init();
        __ReentrancyGuard_init();
        __ERC721Holder_init();

        UNISWAP_V3_POSITION_MANAGER = _uniswapV3PositionManager;
        UNISWAP_V3_FACTORY = _uniswapV3Factory;
        protocolFeeRecipient = _protocolFeeRecipient;
        WETH9 = _weth9;
>>      purpleAddress = _purple;

<...>
    function pauseAndWithdrawToPurple
      (address erc20Address) external onlyOwner nonReentrant validToken(erc20Address)
        BondingCurveData storage curve = curves[erc20Address];
        curve.isPaused = true;
        uint256 ethBalance = curve.ethBalance;
        if (ethBalance > 0) {
            curve.ethBalance = 0;
>>
  (bool ethSent,) = purpleAddress.call{value: ethBalance, gas: 100_000}("");
            if (!ethSent) revert TokenFactory_EthTransferFailed();
            emit EthWithdrawnToPurple(erc20Address, ethBalance);
        }
    }
<...>
    function setPurpleAddress(address _purpleAddress) external onlyOwner {
        emit PurpleAddressUpdated(purpleAddress, _purpleAddress);
>>      purpleAddress = _purpleAddress;
    }
```

Consider checking that the Purple DAO address is not zero in the `pauseAndWithdrawToPurple` function.

[L-05] Deadline doesn't protect against long-pending transactions

In Uniswap v3, the `deadline` parameter represents the Unix time after which the transaction will fail, intended to protect against long-pending transactions and wild swings in prices. However, the deadline set when calling `UNISWAP_V3_POSITION_MANAGER.mint` is `block.timestamp + 15 minutes`. This means that whenever the block proposer decides to include the transaction in a block, it will be valid at that time, as `block.timestamp` will be the current timestamp, rendering the `deadline` ineffective.

```
UNISWAP_V3_POSITION_MANAGER.mint(
            MintParams({
                token0: token0,
                token1: token1,
                fee: poolFee,
                tickLower: tickLower,
                tickUpper: tickUpper,
                amount0Desired: amount0Desired,
                amount1Desired: amount1Desired,
                amount0Min: amount0Min,
                amount1Min: amount1Min,
                recipient: address(this),
                deadline: block.timestamp + 15 minutes // @audit deadline
                // currently set to block timestamp
            })
        );
```

It is recommended to pass the `deadline` as a parameter to the function.

[L-06] moveToAMM can be reverted or lead to fund losses

When moving funds to the Uniswap V3 pool, the contract calculates the `sqrtPriceX96` so that all the ETH balance and token balance can be used to provide liquidity to the pool.

```
function _createPoolAndAddLiquidity(
        address erc20Address,
        uint256 ethBalance,
        uint256 tokenBalance,
        uint16 slippageBips
    ) internal returns (uint256 tokenId) {
        ...
        uint160 sqrtPriceX96 = uint160((Math.sqrt(
          (amount1 * 1e18) / amount0) * _X96) / 1e9);

        // clear eth balance
        curve.ethBalance = 0;

        // Note that The ERC20 balance of this contract is automatically reduced
        // by the transfer
        UNISWAP_V3_POSITION_MANAGER.createAndInitializePoolIfNecessary
        //(token0, token1, poolFee, sqrtPriceX96); // @audit can exist before with dif
        ...
```

However, the pool creation and initialization transaction can be front-run by any account, which can initialize the pool with a very high or low `sqrtPriceX96` value.

- If the maximum slippage tolerated (`slippageBips`) is set, the `moveToAMM` function reverts, causing the funds to be stuck in the TokenFactory contract.
- If the `slippageBips` is not set, the funds moved to the Uniswap V3 pool can be lost due to price manipulation.

It is recommended to check if the pool already exists and the current `sqrtPriceX96`, swap to correct the price before calling `moveToAMM`.

[L-07] The total fee can add up to above 10000

Although there's a check in the `protocolFee` and `creatorFee` to make sure that the fee is not 10000, they can all add up to more than 10000.

```
function setProtocolFeeBips(
      uint256_protocolFeeBips,
      uint256_protocolFeeWithReferralBips
    ) external onlyOwner {
>>      if (_protocolFeeBips > 10000 || _protocolFeeWithReferralBips > 10000) {
            revert TokenFactory_InvalidFeeBips();
        }
        emit ProtocolFeeBipsUpdated(

                        protocolFeeBips, _protocolFeeBips, protocolFeeWithReferralBip
        );
        protocolFeeBips = _protocolFeeBips;
        protocolFeeWithReferralBips = _protocolFeeWithReferralBips;
    }

    function setCreatorFeeBips(
      uint256_creatorFeeBips,
      uint256_creatorFeeWithReferralBips
    ) external onlyOwner {
>>      if (_creatorFeeBips > 10000 || _creatorFeeWithReferralBips > 10000) {
            revert TokenFactory_InvalidFeeBips();
        }
        emit CreatorFeeBipsUpdated(

                        creatorFeeBips, _creatorFeeBips, creatorFeeWithReferralBips,
        );
        creatorFeeBips = _creatorFeeBips;
        creatorFeeWithReferralBips = _creatorFeeWithReferralBips;
    }
```

Either make sure to check the individual fees to be less than a small amount eg 500 (5%), or check in `getFeesFromPrice()` that all three fees do not add up to more than 10000.

Additionally, the fees should not be set that high, so maybe they do not add up to more than 2000 (20%).

```
function getFeesFromPrice(uint256 priceBeforeFees, bool hasReferral)
        public
        view
        returns (uint256 protocolFee, uint256 creatorFee, uint256 referralFee)
    {
        (
          uint256_protocolBips,
          uint256_creatorBips,
          uint256_referralBips
        ) = _getFeeBips(hasReferral
>       require(_protocolBips + _creatorBips + _referralBips < 2000);
        protocolFee = (priceBeforeFees * _protocolBips) / BIPS_DIVISOR;
        creatorFee = (priceBeforeFees * _creatorBips) / BIPS_DIVISOR;
        referralFee = (priceBeforeFees * _referralBips) / BIPS_DIVISOR;
    }
```

[L-08] `x * y = k` invariant not maintained

In constant product AMM with the formula `x * y = k`, a key invariant is that the k value should never decrease after each swap. However, when calculating the ETH amount to buy a specified token amount in `getEthAmountWhenBuyWithExactTokenOutput`, the k value can decrease due to rounding errors, resulting in a loss of liquidity. The calculated `ethAmount` is rounded down.

```
function getEthAmountWhenBuyWithExactTokenOutput
      (address erc20Address, uint256 tokenAmount)
        public
        view
        returns (uint256 ethAmount)
    {
        (uint256 x, uint256 y, uint256 k) = _getCurveDetails(erc20Address);

        if (tokenAmount > x) {
            revert TokenFactory_InvalidTokenAmount();
        }

        uint256 newX = x - tokenAmount;
        uint256 newYScaled = (k * _PRECISION_MULTIPLIER) / newX;

        ethAmount =
        //(newYScaled - y * _PRECISION_MULTIPLIER) / _PRECISION_MULTIPLIER; // @audit
    }
```

Each time the `buyWithReferral` is called, the k value decreases, leading to a lower amount of ETH received for the same amount of tokens sold.

Round up the ethAmount:

```
function getEthAmountWhenBuyWithExactTokenOutput
      (address erc20Address, uint256 tokenAmount)
        public
        view
        returns (uint256 ethAmount)
    {
        (uint256 x, uint256 y, uint256 k) = _getCurveDetails(erc20Address);

        if (tokenAmount > x) {
            revert TokenFactory_InvalidTokenAmount();
        }

        uint256 newX = x - tokenAmount;
        uint256 newYScaled = (k * _PRECISION_MULTIPLIER) / newX;

-       ethAmount =
- (newYScaled - y * _PRECISION_MULTIPLIER) / _PRECISION_MULTIPLIER;
+       ethAmount =
+ (newYScaled - y * _PRECISION_MULTIPLIER) / _PRECISION_MULTIPLIER + 1;
    }
```

Refer to Uniswap V2 calculation: amountIn = (numerator / denominator).add(1);

link

[L-09] Malicious creators can gas-grief users

When buying or selling a token, there is a creator fee. This fee in ETH, is sent directly to the creator fee recipient. Note that there is a gas limit of 100,000.

```
(
          boolcreatorFeeSuccess,

        ) = creatorFeeRecipients[erc20Address].call{value: creatorFee, gas: 100_000}("
        if (!creatorFeeSuccess) {
            revert TokenFactory_CreatorFeeTransferFailed();
        }
```

A malicious creator can set their `creatorFeeRecipient` as a smart contract, and invoke a bunch of random operations (eg loop 1000 times for nothing) just to incur gas and gas grief the users to pay more gas than intended.

Recommend not directly sending the ETH to the creator as they can be malicious and create many issues. Instead, let the creator withdraw the fees

themselves (create a mapping that tracks all the creator fees earned, and add another function to allow the creator to withdraw the ETH that he has earned).