Groupcoin Security Review

Pashov Audit Group

Conducted by: peanuts, ZanyBonzy, kaden

July 5th 2024 - July 7th 2024

Contents

1. About Pashov Audit Group 3

2. Disclaimer 3

3. Introduction 3

4. About Groupcoin 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] Uniswap v3 launch can be enabled without raising enough ETH 9

    [C-02] Tokens can be launched to Uniswap v3 anytime, allowing to drain ETH 10

  8.2. High Findings 12

    [H-01] The same Groupcoin can be launched more than once 12

    [H-02] ETH gets locked in the GroupcoinFactory contract 12

  8.3. Medium Findings 15

    [M-01] Expected totalSupply of 10M tokens not reached after launching to Uniswap v3 15

    [M-02] Users can launch multiple Groupcoins under the same name and ticker 16

    [M-03] Possible sandwich attack when buying a coin 17

    [M-04] Various use of "transfer" opcode to send ETH 18

    [M-05] Fee collector address cannot be set 19

    [M-06] Griefing the launch to UniswapV3 by creating the pool first 20

  8.4. Low Findings 23

[L-01] tokenIdOf will be unreliable for the initial tokenId 23
[L-02] Provided ticks will not work with all pool fees 23
[L-03] Enabling launch can be dossed by selling dust 24
[L-04] Risks due to centralization 24
[L-05] Signatures do not expire and cannot be canceled 25
[L-06] Uniswap v3 launches can be affected by modifying the protocolFee 25

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 **ControlCplusControlV/groupcoin** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Groupcoin

Groupcoin Factory set creates and manages Groupcoin, a custom ERC20 token designed for Telegram groups, with integrated Uniswap V3 liquidity management and a bonding curve pricing model. LiquidityLocker holds Uniswap V3 liquidity positions, allowing the collection and distribution of fees to the token creator and protocol.

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 - ebcf64a54fc9a028ae115425c770fa325c27138d

fixes review commit hash - 5b2a6cc9198f3af50fffa082e64225d5ce91db0d

Scope

The following smart contracts were in scope of the audit:

- `LiquidityLocker`
- `GroupcoinFactory`

7. Executive Summary

Over the course of the security review, peanuts, ZanyBonzy, kaden engaged with Groupcoin to review Groupcoin. In this period of time a total of **16** issues were uncovered.

Protocol Summary

Protocol Name	Groupcoin
Repository	https://github.com/ControlCplusControlV/groupcoin
Date	July 5th 2024 - July 7th 2024
Protocol Type	Bonding curve token sale

Findings Count

Severity	Amount
Critical	2
High	2
Medium	6
Low	6
Total Findings	**16**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Uniswap v3 launch can be enabled without raising enough ETH	Critical	Resolved
[C-02]	Tokens can be launched to Uniswap v3 anytime, allowing to drain ETH	Critical	Resolved
[H-01]	The same Groupcoin can be launched more than once	High	Resolved
[H-02]	ETH gets locked in the GroupcoinFactory contract	High	Resolved
[M-01]	Expected totalSupply of 10M tokens not reached after launching to Uniswap v3	Medium	Resolved
[M-02]	Users can launch multiple Groupcoins under the same name and ticker	Medium	Resolved
[M-03]	Possible sandwich attack when buying a coin	Medium	Acknowledged
[M-04]	Various use of "transfer" opcode to send ETH	Medium	Resolved
[M-05]	Fee collector address cannot be set	Medium	Resolved
[M-06]	Griefing the launch to UniswapV3 by creating the pool first	Medium	Acknowledged
[L-01]	tokenIdOf will be unreliable for the initial tokenId	Low	Resolved
[L-02]	Provided ticks will not work with all pool fees	Low	Resolved

[L-03]	Enabling launch can be dossed by selling dust	Low	Acknowledged
[L-04]	Risks due to centralization	Low	Acknowledged
[L-05]	Signatures do not expire and cannot be canceled	Low	Acknowledged
[L-06]	Uniswap v3 launches can be affected by modifying the protocolFee	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Uniswap v3 launch can be enabled without raising enough ETH

Severity

Impact: High

Likelihood: High

Description

`enableUniswapV3Launch` can only be executed if `unitPrice >= config.thresholdPrice`, which is implicitly intended to enforce that at least `config.token1Amount` ETH has been paid to purchase the given token. It's important that enough ETH has been paid so that we have enough to supply liquidity. However, users only need to buy ~8,200,000 tokens for the `unitPrice` to meet the `thresholdPrice`, which is ~155,000 tokens less than we need to sell to have sufficient ETH. We can prove this with the following test:

```
function testLaunchToUniswapV3() public {
    bytes32 message = keccak256(abi.encodePacked(address
      (this), "Telegram WIF Coin", "WIF"));
    (uint8 v, bytes32 r, bytes32 s) = vm.sign
      (0xbeef, message.toEthSignedMessageHash());
    bytes memory signature = abi.encodePacked(r, s, v);

    (bytes32 salt,) = factory.generateSalt(address
      (this), "Telegram WIF Coin", "WIF");
    uint256 newToken = factory.launchGroupCoin
      ("Telegram WIF Coin", "WIF", signature, salt, 0);
    uint256 price = factory.getBuyPriceAfterFee(newToken, 8_357_142);

    vm.deal(User01, price);
    vm.startPrank(User01);

    factory.buy{ value: price }(newToken, 8_200_000);

    factory.enableUniswapV3Launch(newToken);
}
```

Since it's not possible to `buy` more tokens after a launch is committed, the only way to successfully launch Uniswap v3 after an early commit like this is to use additional ETH in the contract raised by other tokens, if available, making the contract insolvent.

Recommendations

The best solution to this is likely to gate `enableUniswapV3Launch` according to the `totalSupply` of the token such that we can be certain that enough ETH has been paid to buy the tokens.

[C-02] Tokens can be launched to Uniswap v3 anytime, allowing to drain ETH

Severity

Impact: High

Likelihood: High

Description

In `GroupcoinFactory`, we intend to gate launching to Uniswap v3 with a two-step approach. First, the user calls `enableUniswapV3Launch`, which only succeeds if the current buy price for one token is at least the `config.thresholdPrice`. Then, it should only be possible to execute `launchGroupCoinToUniswapV3` if we had successfully executed the previous step. However, due to a logical error, it's possible to execute `launchGroupCoinToUniswapV3` without first executing `enableUniswapV3Launch`.

In `launchGroupCoinToUniswapV3`, the following check is intended to only pass if we have enabled the launch in a previous block:

```
if (block.number <= commitLaunch[tokenId]) {
    revert CantLaunchPoolYet();
}
```

However, since the unset value for `commitLaunch[tokenId]` is 0, this will pass for `tokenId`'s which has not been enabled. We can prove this with the following test:

```
function testLaunchToUniswapV3() public {
    bytes32 message = keccak256(abi.encodePacked(address
      (this), "Telegram WIF Coin", "WIF"));
    (uint8 v, bytes32 r, bytes32 s) = vm.sign
      (0xbeef, message.toEthSignedMessageHash());
    bytes memory signature = abi.encodePacked(r, s, v);

    (bytes32 salt,) = factory.generateSalt(address
      (this), "Telegram WIF Coin", "WIF");
    uint256 newToken = factory.launchGroupCoin
      ("Telegram WIF Coin", "WIF", signature, salt, 0);

    // @audit deal ETH so we can add as liquidity
    vm.deal(address(factory), 2 ether);

    factory.launchGroupCoinToUniswapV3(newToken);
}
```

As we can see above, it's possible to launch a token to Uniswap v3 without first enabling it, which means that it doesn't have to meet the intended criteria to be launched. The impact of this is that attackers can atomically buy tokens at the start of the bonding curve for cheap, then launch the token to Uniswap v3, taking ETH from the contract which would be allocated for other tokens' liquidity positions, and finally sell their cheap tokens at the inflated price.

Recommendations

We must ensure that `commitLaunch[tokenId]` is not unset, e.g.:

```
if (block.number <= commitLaunch[tokenId] || commitLaunch[tokenId] == 0) {
    revert CantLaunchPoolYet();
}
```

8.2. High Findings

[H-01] The same Groupcoin can be launched more than once

Severity

Impact: High

Likelihood: Medium

Description

In `launchGroupCoinToUniswapV3`, there is no validation that the Groupcoin hasn't already been launched. Under normal circumstances, this is not a problem because it's not possible to create multiple pools with the same token pair and fee. However, it's possible for the owner to change the fee at any time via `setUniswapConfig`. As a result, if the owner changes the fee, any Groupcoin that was previously launched with the old fee can now be launched again.

Re-launching a Groupcoin causes additional tokens to be minted, causing its `totalSupply` to exceed the intended 10m token maximum. Additionally, since ETH is also provided to the liquidity position and the ETH received from initially selling the Groupcoin has already been used in the previous launch, this will take ETH received from other Groupcoin sales, preventing other Groupcoins from being launched.

Recommendations

Trigger a revert at the top of `launchGroupCoinToUniswapV3` if `launched[tokenId] = true`.

[H-02] ETH gets locked in the `GroupcoinFactory` contract

Severity

Impact: Medium

Likelihood: High

Description

When we launch a Groupcoin to Uniswap v3, we intend to provide all the ETH received from sales as liquidity in the newly created Uniswap v3 pool. However, as we will see below, some ETH may not get wrapped and added to the liquidity position, causing it to be permanently locked in the `GroupcoinFactory` contract.

In `launchGroupCoinToUniswapV3`, we wrap a fixed amount of ETH, as determined by `config.token1Amount`, which is later deposited as liquidity to the created Uniswap v3 pool:

```
IWETH9(WETH).deposit{ value: config.token1Amount }();
```

The `config.thresholdPrice` is reached before we exceed the threshold amount of tokens allowed to be bought. We can prove this by modifying the `testLaunchToUniswapV3` function to buy fewer tokens prior to launching, which works with >2000 fewer tokens:

```
function testLaunchToUniswapV3() public {
    bytes32 message = keccak256(abi.encodePacked(address
      (this), "Telegram WIF Coin", "WIF"));
    (uint8 v, bytes32 r, bytes32 s) = vm.sign
      (0xbeef, message.toEthSignedMessageHash());
    bytes memory signature = abi.encodePacked(r, s, v);

    (bytes32 salt,) = factory.generateSalt(address
      (this), "Telegram WIF Coin", "WIF");
    uint256 newToken = factory.launchGroupCoin
      ("Telegram WIF Coin", "WIF", signature, salt, 0);
    uint256 price = factory.getBuyPriceAfterFee(newToken, 8_357_142);

    vm.deal(User01, price);
    vm.startPrank(User01);

    // @audit we modify the buy amount to be 2110 tokens less and launch still
    // succeeds
    factory.buy{ value: price }(newToken, 8_355_000);

    factory.enableUniswapV3Launch(newToken);
    vm.roll(block.number + 1);
    factory.launchGroupCoinToUniswapV3(newToken);
}
```

This also proves that the fixed amount of ETH that we wrap is less than the maximum amount of ETH which can be received from more tokens being bought because we otherwise wouldn't have enough ETH available to launch.

Since the amount of ETH wrapped and deposited to the pool is less than the maximum amount of ETH that can be received from more tokens being bought, it's possible for some ETH to be left in the contract. Because we don't have a way to withdraw excess ETH from the contract, this excess ETH is permanently locked in the `GroupcoinFactory`. We can prove this with the following modified version of `testLaunchToUniswapV3`:

```
function testLaunchToUniswapV3() public {
    bytes32 message = keccak256(abi.encodePacked(address
      (this), "Telegram WIF Coin", "WIF"));
    (uint8 v, bytes32 r, bytes32 s) = vm.sign
      (0xbeef, message.toEthSignedMessageHash());
    bytes memory signature = abi.encodePacked(r, s, v);

    (bytes32 salt,) = factory.generateSalt(address
      (this), "Telegram WIF Coin", "WIF");
    uint256 newToken = factory.launchGroupCoin
      ("Telegram WIF Coin", "WIF", signature, salt, 0);
    uint256 price = factory.getBuyPriceAfterFee(newToken, 8_357_142);

    vm.deal(User01, price);
    vm.startPrank(User01);

    factory.buy{ value: price }(newToken, 8_357_110);

    factory.enableUniswapV3Launch(newToken);

    vm.roll(block.number + 1);

    factory.launchGroupCoinToUniswapV3(newToken);

    uint256 ethBalance = address(factory).balance;
    assertEq(ethBalance, 0);
}
```

As we can see above, if we buy 8,357,110 tokens, we end up with ~0.001616692 ETH left in the `GroupcoinFactory` contract.

Recommendations

Include an `onlyOwner` function to withdraw ETH left in the contract. Note that this assumes that the `owner` of the `GroupcoinFactory` is a trusted actor.

8.3. Medium Findings

[M-01] Expected `totalSupply` of 10M tokens not reached after launching to Uniswap v3

Severity

Impact: Low

Likelihood: High

Description

The expected `totalSupply` of Groupcoin after launching to Uniswap v3 is 10,000,000. However, we can validate from testing that we never actually quite reach 10,000,000. We can see from the result of the below test that even if we buy the maximum amount of tokens, we will still have a `totalSupply` of less than 10M:

```
function testLaunchToUniswapV3() public {
    bytes32 message = keccak256(abi.encodePacked(address
      (this), "Telegram WIF Coin", "WIF"));
    (uint8 v, bytes32 r, bytes32 s) = vm.sign
      (0xbeef, message.toEthSignedMessageHash());
    bytes memory signature = abi.encodePacked(r, s, v);

    (bytes32 salt,) = factory.generateSalt(address
      (this), "Telegram WIF Coin", "WIF");
    uint256 newToken = factory.launchGroupCoin
      ("Telegram WIF Coin", "WIF", signature, salt, 0);
    uint256 price = factory.getBuyPriceAfterFee(newToken, 8_357_142);

    vm.deal(User01, price);
    vm.startPrank(User01);

    factory.buy{ value: price }(newToken, 8_357_111);

    factory.enableUniswapV3Launch(newToken);

    vm.roll(block.number + 1);

    factory.launchGroupCoinToUniswapV3(newToken);

    Groupcoin coin = factory.addressOf(newToken);
    assertEq(coin.totalSupply(), 10_000_000 ether);
}
```

Furthermore, we can buy >2000 tokens less than the maximum amount since we will reach the `config.thresholdPrice` before then, allowing us to launch Uniswap v3 regardless:

```
function testLaunchToUniswapV3() public {
    bytes32 message = keccak256(abi.encodePacked(address
      (this), "Telegram WIF Coin", "WIF"));
    (uint8 v, bytes32 r, bytes32 s) = vm.sign
      (0xbeef, message.toEthSignedMessageHash());
    bytes memory signature = abi.encodePacked(r, s, v);

    (bytes32 salt,) = factory.generateSalt(address
      (this), "Telegram WIF Coin", "WIF");
    uint256 newToken = factory.launchGroupCoin
      ("Telegram WIF Coin", "WIF", signature, salt, 0);
    uint256 price = factory.getBuyPriceAfterFee(newToken, 8_357_142);

    vm.deal(User01, price);
    vm.startPrank(User01);

    // @audit we buy less tokens and yet we can still launch
    factory.buy{ value: price }(newToken, 8_355_000);

    factory.enableUniswapV3Launch(newToken);

    vm.roll(block.number + 1);

    factory.launchGroupCoinToUniswapV3(newToken);

    Groupcoin coin = factory.addressOf(newToken);
    assertEq(coin.totalSupply(), 10_000_000 ether);
}
```

In the above test, we only end up with a `totalSupply` of ~9,997,888 Groupcoins.

Recommendations

In `launchGroupCoinToUniswapV3`, instead of minting a hardcoded amount of Groupcoins, we should mint `10_000_000 ether - token.totalSupply()`, e.g.:

```
token.mint(address(this), 10_000_000 ether - token.totalSupply());
```

[M-02] Users can launch multiple Groupcoins under the same name and ticker

Severity

Impact: Medium

Likelihood: Medium

Description

In `launchGroupCoin`, a hash of the `msg.sender`, `name`, and `ticker` is taken and converted to a signed message hash where it's compared to the provided `signature` where we revert if the recovered address doesn't match the `signer`. This is done so that users can only launch Groupcoins if the `signer` has allowed for them to launch one with the given `name` and `ticker`. However, the signature logic doesn't contain a nonce, which allows the user to launch as many Groupcoins as they want under the allowed `name` and `ticker` by providing different valid `salt`'s.

Recommendations

Include a `nonce` in the signed message hash which references a state variable that gets incremented after using the signature. Furthermore, the `chainid` should also be included in the message hash so that the Groupcoin can only be launched on the intended chain in case of a chain split.

[M-03] Possible sandwich attack when buying a coin

Severity

Impact: High

Likelihood: Low

Description

When buying a coin through `buy()` in GroupcoinFactory.sol, the buyer will attach some `msg.value` to the function in order to buy a certain number of coins. In `buy()`, there is no slippage protection, so a malicious user can buy

For example,

- Alice wants to buy 1000 Acoin. She is the first buyer and she calculates that she needs about 3434501 wei of ETH to buy (amount calculated from the `getPrice()` function. Alice knows that the protocol will refund any excess ETH that she sends in, so she simply passes in 1e18 worth of ETH.
- Bob sees that transaction and frontruns her, by buying 1,000,000 Acoin for about 3429360425241769 wei. Now, Alice's transaction goes through, and `totalSupply()` of Acoin becomes 1 million instead of 0.
- Alice 1000 Acoins is now 10298367632031 wei of ETH, and the transfer goes through because she attached 1e18 worth of ETH.

Recommendations

To protect the buyers, add another parameter `maxPrice` to make sure that the buyer is willing to pay x amount of tokens until it reaches the max price. In Alice's case, her max price would be 4000000 wei, so her transaction will be reverted if the coin is any price above that.

Sample example:

```
function buy
      (uint256 tokenId, uint256 amount, uint256 maxPrice) public payable {
        ...
        if (msg.value < price + protocolFeeTaken) {
            revert NotEnough();
        }
        if(maxPrice != 0){
        require(maxPrice > price + protocolFeeTaken, "Price too expensive")
        }
```

If the maxPrice is not set, then it is assumed that the buyer accepts that there is no slippage protection.

[M-04] Various use of "transfer" opcode to send ETH

Severity

Impact: Medium

Likelihood: Medium

Description

In GroupcoinFactory.sol, when buying and selling tokens, the `.transfer` opcode is used to handle ETH transfer, it does this by forwarding a fixed amount of 2300 gas. This is dangerous for two reasons:

1. Gas costs of EVM instructions may change significantly during hard forks which may previously assumed fixed gas costs. EIP 1884 as an example, broke several existing smart contracts due to a cost increase of the SLOAD instruction.
2. If the recipient is a contract or a multisig safe, with a `receive`/`fallback` function which requires >2300 gas, e.g safes that execute extra logic in the `receive`/`fallback` function, the transfer function will always fail for them due to out of gas errors.

In the `buy` function, sending ETH to the `feeTo` address or to `msg.sender` will fail if any of the above conditions hold, and as such users will not be able to buy tokens, or will not be able to receive a refund.

```
payable(feeTo).transfer(protocolFeeTaken);

        uint256 owed = price + protocolFeeTaken;
        if (msg.value > owed) {
            payable(msg.sender).transfer(msg.value - owed);
        }
```

In the `sell` function, sending ETH to the `feeTo` address or to `msg.sender` will fail if any of the above conditions hold, and as such, users will not be able to sell their tokens.

```
payable(feeTo).transfer(protocolFeeTaken);
        payable(msg.sender).transfer(price);
```

Recommendations

Use the ".call" opcode instead, and add a non reentrant modifier to the `buy` and `sell` functions.

[M-05] Fee collector address cannot be set

Severity

Impact: Low

Likelihood: High

Description

To claim fees, the creator or the collector can call the `claimFees` function.

```
function claimFees(uint256 id) external {
        address creator = creatorOf[id];
        if (!canCollectFees[msg.sender] && msg.sender != creator) {
            revert NoAuthorization();
        }
```

To set the fee collector, a call has to be made from the GroupcoinFactory, as can be seen from the check. The function is however missing in the GroupcoinFactory contract. As a result, fee collectors cannot be set, hence, they cannot claim fees.

```
function delegateFeeCollection(address collector, bool isDelegated) public {
        if (msg.sender != factory) {
            revert NoAuthorization();
        }

        canCollectFees[collector] = isDelegated;
    }
```

Recommendations

Introduce a function in GroupcoinFactory.sol that allows creators to set their fee collectors.

[M-06] Griefing the launch to UniswapV3 by creating the pool first

Severity

Impact: Medium

Likelihood: Medium

Description

Once x amount of coins are bought and the value of the coin reaches a certain threshold, `launchGroupCoinToUniswapV3` can be called. This will transition the trading of coins to a different bonding curve.

In `launchGroupCoinToUniswapV3()`, the function will call `uniswapV3Factory.createPool` without checking the existence of the pool.

```
address pool = uniswapV3Factory.createPool(address
          (token), WETH, config.fee);
        IUniswapV3Pool(pool).initialize(config.sqrtPriceX96);
```

In UniswapV3Factory.createPool, it only allows the pool to be created once:

```
function createPool(
        address tokenA,
        address tokenB,
        uint24 fee
    ) external override noDelegateCall returns (address pool) {
        require(tokenA != tokenB);
        (address token0, address token1) = tokenA < tokenB ?
          (tokenA, tokenB) : (tokenB, tokenA);
        require(token0 != address(0));
        int24 tickSpacing = feeAmountTickSpacing[fee];
        require(tickSpacing != 0);
 >      require(getPool[token0][token1][fee] == address(0));
        pool = deploy(address(this), token0, token1, fee, tickSpacing);
 >      getPool[token0][token1][fee] = pool;
        // populate mapping in the reverse direction, deliberate choice to avoid
        // the cost of comparing addresses
        getPool[token1][token0][fee] = pool;
        emit PoolCreated(token0, token1, fee, tickSpacing, pool);
    }
```

A malicious user can buy some tokens and call `createPool` directly in the UniswapV3Factory, with tokenA as the coin and tokenB as WETH and the fee according to the protocol's config.

Then, when `launchGroupCoinToUniswapV3()` is called, the function will revert as the pool has already been created.

Recommendations

There is a function from another protocol that checks whether the pool has already been created and simply calls initialize if so.

A potential solution looks something like this:

```
function _initUniV3PoolIfNecessary(
   PoolAddress.PoolKeymemorypoolKey,
   uint160sqrtPriceX96
 ) internal returns (address pool
        pool = IUniswapV3Factory(UNIV3_FACTORY).getPool
          (poolKey.token0, poolKey.token1, poolKey.fee);
        if (pool == address(0)) {
            pool = IUniswapV3Factory(UNIV3_FACTORY).createPool
              (poolKey.token0, poolKey.token1, poolKey.fee);
            IUniswapV3Pool(pool).initialize(sqrtPriceX96);
        } else {
            (uint160 sqrtPriceX96Existing, , , , , , ) = IUniswapV3Pool
              (pool).slot0();
            if (sqrtPriceX96Existing == 0) {
                IUniswapV3Pool(pool).initialize(sqrtPriceX96);
            } else {
                require(sqrtPriceX96Existing == sqrtPriceX96, "UV3P");
            }
        }
    }
```

8.4. Low Findings

[L-01] `tokenIdOf` will be unreliable for the initial `tokenId`

Since `launchGroupCoin` sets the `tokenId` as the current value of `maxGroupcoin`, the initial `tokenId` in `GroupcoinFactory` is 0. When setting the `tokenIdOf` mapping value for the initial `tokenId`, this will set `tokenIdOf[initialCoin] = 0`. This can cause a great deal of confusion because the default value of unset mapping keys is 0, thus simply by referencing `tokenIdOf`, it will be impossible to ascertain whether the reference coin actually exists.

We can resolve this by initializing `maxGroupcoin` to 1. This will also save gas on the first coin launch since incrementing `maxGroupcoin` from zero to non-zero is much more expensive than incrementing from non-zero to non-zero.

[L-02] Provided ticks will not work with all pool fees

Uniswap v3 pools have different tick spacing sets depending on the fee used. For a fee of `10000` which is the initial value used in `GroupcoinFactory`, we have a tick spacing of 200, which means that the provided ticks used must be a multiple of 200. When we `mint` a liquidity position in `launchGroupCoinToUniswapV3`, we provide a hardcoded tick range of `[-887_200, 887_200]`. These ticks are multiples of 200 and thus can be used. However, if we were to use a fee of `3000`, the tick spacing would be 60, which our ticks are not evenly divisible by. As a result, `launchGroupCoinToUniswapV3` would always revert if `config.fee = 3000`.

To resolve this, we can instead use a tick range of `[-887_400, 887_400]` which should work with any fees' tick spacing while providing roughly the same range.

[L-03] Enabling launch can be dossed by selling dust

Before the token can be enabled for launch, malicious users can sell small token amounts to continuously push the price below the threshold. In an extreme case, frontrunning the `enableUniswapV3Launch` function to sell small token amounts, just enough to keep the price below the threshold without losing substantial amounts in sales/fees (note that buying those small amounts will be a bit inconvenient for other users), dossing the token launch.

```
function enableUniswapV3Launch(uint256 tokenId) public {
        uint256 unitPrice = getBuyPriceAfterFee(tokenId, 1);

        if (unitPrice < config.thresholdPrice) {
            revert NotAtThreshold();
        }

        commitLaunch[tokenId] = block.number;

        emit UniswapLaunchStaged(tokenId, block.number);
    }
```

Recommend introducing a minimum amount of tokens that can be sold.

[L-04] Risks due to centralization

Admin-protected functions have the potential to negatively affect users if the admin gets compromised.

1. In GroupcoinFactory.sol, `setProtocolFee` should have a cap, so as not to negatively grieve users when buying and selling. Setting very high can potentially make selling the token impossible, creating a potential honeypot.

```
function setProtocolFee(uint256 newProtocolFee) public onlyOwner {
        protocolFee = newProtocolFee;

        emit ProtocolFeeChanged(newProtocolFee);
    }
```

2. In LiquidityLocker.sol, the same cap should be implemented to prevent creators from losing pool fees when they claim.

```
function setProtocolFee(uint256 newProtocolFee) public onlyOwner {
        protocolFee = newProtocolFee;
    }
```

[L-05] Signatures do not expire and cannot be canceled

The provided signature doesn't have a deadline parameter or a nonce, so the signatures can be held indefinitely and cannot be canceled, at least until the `signer` address is changed. If the owner decides that a certain user is no longer deserving of the signature (the user is malicious or something), the only way to prevent such a user from deploying is to set a new `signer` which will involve inconveniencing other honest users and if the old signer is reinstated, the signatures will become valid..

```
function launchGroupCoin(
        string calldata name,
        string calldata ticker,
        bytes calldata signature,
        bytes32 salt,
        uint256 amount
    )
        public
        payable
        returns (uint256 tokenId)
    {
        bytes32 hash = keccak256(abi.encodePacked(msg.sender, name, ticker));
        bytes32 signedHash = hash.toEthSignedMessageHash();
        address approved = ECDSA.recover(signedHash, signature);
        if (approved != signer) {
            revert NotApproved();
        }
```

Recommend introducing a deadline parameter, and checking for expiration time. A nonce parameter and an owner-protected function to use nonce will also help to cancel a signature.

[L-06] Uniswap v3 launches can be affected by modifying the `protocolFee`

In `enableUniswapV3Launch`, we check whether the given token is ready to be launched to Uniswap v3 according to if the buy price including the protocol fee has reached the `config.thresholdPrice`:

```
uint256 unitPrice = getBuyPriceAfterFee(tokenId, 1);

if (unitPrice < config.thresholdPrice) {
    revert NotAtThreshold();
}
```

```
function getBuyPriceAfterFee
  (uint256 tokenId, uint256 amount) public view returns (uint256 price) {
    price = getBuyPrice(tokenId, amount);
    price += price * protocolFee / 1 ether;
}
```

Since we're including the `protocolFee` in this logic, and the `owner` can change the `protocolFee` at any time, modifications to the `protocolFee` could result in unexpected effects. In the case that the `protocolFee` is increased, the returned `unitPrice` will exceed the `config.thresholdPrice` earlier than it should, allowing a Uniswap v3 launch to be committed before enough ETH is raised for the liquidity position, making the system insolvent.

Use a `thresholdPrice` which references and compares to the unit buy price, not including the `protocolFee`.