CD SECURITY

AUDIT REPORT

Molly Token
December 2023

Introduction

A time-boxed security review of the **Molly** token was done by **ddimitrov22** and **chrisdior4**, with a focus on the security aspects of the application's implementation.

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.

About **Molly**

Molly is an ERC20 token with a modified transfer function. There are two distinct buyer groups: angel and private.

To claim allocated tokens, individuals must undergo a verification process involving off-chain KYC. Upon successful validation, a signature from project's team private key is acquired, and users are prompted to mark their verification status within the contract. Merkle proofs are employed to verify the claimable amount, generated off-chain.

Fees are integrated into this system. Angel participants begin at a 90% fee, + the public fee, gradually diminishing over 120 days until it reaches 0 plus the public fee. Similarly, private participants start at 80% fee, reducing evenly over 90 days until it reaches 0 plus the public fee.

Severity classification

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

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - f2167abb43bca9ecfecfb7ab2bad95668f80f86a**

Scope

The following smart contracts were in scope of the audit:

- `Molly.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 5 issues
- Medium: 1 issues
- Low: 2 issues
- Informational: 6 issues

Findings Summary

ID	Title	Severity
[C-01]	The `verifyUser` method is flawed and it will not work	Critical
[C-02]	Users will not be able to claim tokens during `AngelSale` and `PrivateSale`	Critical
[H-01]	Lack of slippage control can lead to sandwich attacks	High
[H-02]	Deadline check is not effective	High
[H-03]	The `_transfer` function will not work for normal users	High
[M-01]	Insufficient input validation	Medium
[L-01]	Use two-step ownership transfer approach	Low
[L-02]	Discrepancy between comment and code	Low
[I-01]	Using `SafeMath` when compiler is ^0.8.0	Informational
[I-02]	Redundant modifier	Informational
[I-03]	Unused code	Informational
[I-04]	Prefer Solidity Custom Errors over `require` statements with strings	Informational
[I-05]	Missing event emissions in state changing methods	Informational
[I-06]	Use a stable pragma statement	Informational

Detailed Findings

[C-01] The `verifyUser` method is flawed and it will not work

Severity

Impact: High, as users should be verified to claim the allocated tokens.

Likelihood: High, as no users will be verified ever.

Description

There are several problems with the `verifyUser` function:

```
    function verifyUser(bytes memory _data, bytes memory _signature)
external {
        bytes32 _hash = ECDSA.toEthSignedMessageHash(keccak256(_data));

        require(
            ECDSA.recover(_hash, _signature) == signer,
            "Invalid signature"
        );

        isVerified[msg.sender] = true;
    }
```

1. The first is the use of `ECDSA.verify` method to validate the signature.

```
        require(
            ECDSA.recover(_hash, _signature) == signer,
            "Invalid signature"
        );
```

The problem is that it will compare it to the `signer` which is set to `address(0xdead)` and there is no way to change it. The `signer` is set in the constructor by calling the internal `_setSigner` function and there is no way to call it again and change it to another `signer`:

```
    constructor() ERC20("Molly", "MOLY") {
        ...
        _setSigner(address(0xdead));
    }
```

This will lead to the `require` statement always reverting and returning `"Invalid signature"`.

2. The second problem is that even if the above problem is fixed, anyone can front run the call with the same input parameters and be verified. For example, user Bob calls the function with valid `bytes memory _data, bytes memory _signature` parameters which will pass the require statement and he will become verified. However, while the call is in the mempool, malicious user Alice sees the

transaction and calls the function with the same input parameters as Bob but with higher gas price. This will result in Alice being a verified user even if she is not supposed to be.

3. The third problem is that the function calls the `ECDSA.toEthSignedMessageHash` method. However, in the latest OpenZeppelin version of the libraries, the `toEthSignedMessageHash` method is part of the `MessageHashUtils` library which should be imported as well, as it is best security practice to use the latest versions of OZ libraries as they are constantly updated and optimized.

Recommendations

Consider comparing the `ECDSA.recover` signature to a valid `signer`. Also, include the `msg.sender` or `nonce` as part of the validation as this will make the validation specific for each user. Finally, update the OZ libraries to the latest version and import the above-mentioned library.

[C-02] Users will not be able to claim tokens during `AngelSale` and `PrivateSale`

Severity

Impact: High because none of the users will be able to claim any tokens.

Likelihood: High because the `claim` functions will be DoSed.

Description

The `claimAngelSale` and `claimPrivateSale` functions are designed to allow each specific group of users to claim the allocated tokens for them. The `MerkleProof.verify` method is used to verify the `_merkleProof` provided by the user and check whether he should be able to claim tokens.

```
    function claimAngelSale(
        uint256 _amount,
        bytes32[] calldata _merkleProof
    ) external {
        require(isVerified[msg.sender], "Not verified");
        bytes32 leaf = keccak256(abi.encodePacked((msg.sender), _amount));
        require(
            MerkleProof.verify(_merkleProof, merkleRoot, leaf),
            "Invalid proof!"
        );
        require(!AngelClaimed[msg.sender], "Already claimed");
        _transfer(address(this), msg.sender, _amount);
        AngelClaimed[msg.sender] = true;
    }
```

However, the `merkleRoot` which is used to be compared against the `_merkleRoof` is never set and will be with default value `bytes32(0)`. The same applies to the `privateMerkleRoot`:

```
    bytes32 public merkleRoot;

    bytes32 public privateMerkleRoot;
```

Even though the `_merkleProof` is generated off-chain and it might be valid, the function will still revert with `"Invalid proof!"` as it will compare it to the default value. This will make it impossible for the users to claim any amount of tokens as there is no method to set the `merkleRoot` and `privateMerkleRoot` once the contract is deployed.

Recommendations

Either set the two `merkleRoot` variables inside the constructor or create functions to set them later but this will require additional check inside the `claim` function to check if the `merkleRoot` is set.

[H-01] Lack of slippage control can lead to sandwich attacks

Severity

Impact: High, as this will lead to loss of funds for users

Likelihood: Medium, since MEV is very prominent, the chance of that happening is pretty high

Description

The `amountOutMin` parameter in `swapExactTokensForETHSupportingFeeOnTransferTokens` is hard coded to 0 in `swapTokensForEth()`:

```
 function swapTokensForEth(uint256 tokenAmount) private {
        ...
       // make the swap

uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            tokenAmount,
            0, // accept any amount of ETH //@audit no slippage protection
            path,
            address(this),
            block.timestamp
        );
    }
```

This basically allows for 100% slippage as the call agrees to receive 0 amount of ETH for the swap. This can be done through a sandwich attack. The same applies to the `openTrade` function:

```
function openTrade() external payable onlyOwner {
        _approve(address(this), address(uniswapV2Router), totalSupply());
        uniswapV2Router.addLiquidityETH{value: address(this).balance}(
            address(this),
            balanceOf(address(this)),
            0, //uint amountTokenMin, //@audit both inputs here should be
user supplied parameters, not hardcoded to 0
            0, //uint amountETHMin,
            owner(),
            block.timestamp
        );
```

This is a very easy target for MEV and bots to do a flash loan sandwich attack and can be done on every call if the trade transaction goes through a public mempool.

Recommendations

The best solution to this problem is to add an input parameter instead of hardcoding 0. The `amountOutMin` can be calculated off-chain and agreed upon by the user and can be passed to the call. This will protect the calls from sandwich attacks.

[H-02] Deadline check is not effective

Impact: High, because the transaction might be left hanging in the mempool and be executed way later than the user wanted at a possibly worse price

Likelihood: Medium, because there is a great chance that the user won't adjust the gas price to be lucrative for the validators to include its transaction fast

The deadline parameter in `swapExactTokensForETHSupportingFeeOnTransferTokens()` and `addLiquidityETH()` which are called in `swapTokensForEth()` and `openTrade()` is hardcoded to `block.timestamp`.

Example in `openTrade()`:

```
function openTrade() external payable onlyOwner {
        _approve(address(this), address(uniswapV2Router), totalSupply());
        uniswapV2Router.addLiquidityETH{value: address(this).balance}(
            address(this),
            balanceOf(address(this)),
            0, //uint amountTokenMin,
            0, //uint amountETHMin,
            owner(),
            block.timestamp
        );
```

The `addLiquidityETH()` in `UniswapV2Router02` contract:

```
function addLiquidityETH(
        address token,
        uint amountTokenDesired,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external virtual override payable ensure(deadline) returns (uint
amountToken, uint amountETH, uint liquidity)
{
```

The `deadline` parameter enforces a time limit by which the transaction must be executed otherwise it will revert.

Let's take a look at a modifier that is present in the functions you are calling in `UniswapV2Router02` contract:

```
modifier ensure(uint deadline) {
        require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
        _;
    }
```

Now when the `deadline` is hardcoded as `block.timestamp`, the transaction will not revert because the require statement will always be fulfilled by block.timestamp == block.timestamp.

If a user chooses a transaction fee that is too low for miners to be interested in including the transaction in a block, the transaction stays pending in the mempool for extended periods, which could be hours, days, weeks, or even longer.

This could lead to users getting a worse price because a validator can just hold onto the transaction.

Recommendations

Protocols should let users who interact with AMMs set expiration deadlines. Without this, there's a risk of a serious loss of funds for anyone starting a swap, especially if there's no slippage parameter.

Use a user-supplied deadline instead of `block.timestamp`.

[H-03] The `_transfer` function will not work for normal users

Severity

Impact: High, as the users will not be able to transfer any tokens

Likelihood: Medium, as this will happen after a given period of time

Description

The `_transfer` function performs different checks to see if there are any constraints like `limitsInEffect`, `canSwap`, etc. It also checks if the sender or the receiver is excluded from fees to decide if any fees should be applied.

```
        if (takeFee) {
            // on sell
            if (automatedMarketMakerPairs[to] && sellFees > 0) {
                if (isAngelBuyer[from]) {
                    uint256 currentFee = getCurrentAngelFee();

                    fees = amount.mul(currentFee + sellFees).div(100);
                } else if (isPrivateSaleBuyer[from]) {
                    uint256 currentFee = getCurrentFee();
                    fees = amount.mul(currentFee + sellFees).div(100);
                } else {
                    fees = amount.mul(sellFees).div(100);
                }
            }
            // on buy
            else if (automatedMarketMakerPairs[from] && buyFees > 0) {
                if (isAngelBuyer[to]) {
                    uint256 currentFee = getCurrentAngelFee();

                    fees = amount.mul(currentFee + buyFees).div(100);
                } else if (isPrivateSaleBuyer[to]) {
                    uint256 currentFee = getCurrentFee();

                    fees = amount.mul(currentFee + buyFees).div(100);
                } else {
                    fees = amount.mul(buyFees).div(100);
                }
            }
```

The fees are designed to decrease over time and reach 0 after a specific period of time - 90 days for private sale and 120 days for Angel sale. The `getCurrentFee` and `getCurrentAngelFee` are called to calculate the right amount of fees.

```
    function getCurrentFee() public view returns (uint256) {
        uint256 daysPassed = (block.timestamp - startDate) / 60 / 60 / 24;
        uint256 currentFee = initialFee - (daysPassed * dailyDecrease);
        if (currentFee < 0) {
            currentFee = 0;
        }
        return currentFee;
    }
```

The problem is that the `currentFee` is stored in `uint256` variable which will always revert when `daysPassed * dailyDecrease > initialFee` which will happen after 90 days. This is because of the solidity compiler that will check for overflow and underflow errors and will revert. Even though there is an `if` statement to set the `currentFee` to 0, the call will revert before that as the `uint256` can never be a negative number. The same applies to the `getCurrentAngelFee` where the same thing will happen but after 120 days.

Recommendations

Store the `currentFee` in `int256` to allow it to be a negative number before setting it to 0.

[M-01] Insufficient input validation

Severity

Impact: Medium, because a protocol can be broken and the code could give false calculations

Likelihood: Medium, as it can be gamed but it needs a compromised / malicious owner

Description

There are a couple of instances where the functions input params are missing a proper validation. It's okay that these functions are only callable by the owner but if we have a malicious or compromised owner there might be a serious problem.

Example is:

```
function updateFees(uint256 _fee) external onlyOwner {
        buyFees = _fee;
        sellFees = _fee;
    }
```

Make the same validations for the following functions as well:

- `updateSwapTokensAtAmount()`
- `updateBuyFees()`
- `updateSellFees()`

Recommendations

Add sensible constraints and validations for all user input mentioned above. Example for `updateFees()`:

```
require(_fee <= 100 && _fee > 0, "New fee is out of boundaries");
```

[L-01] Use two-step ownership transfer approach

The `owner` role is crucial for the protocol as there are a lot of functions with the `onlyOwner` modifier. Make sure to use a two-step ownership transfer approach by using `Ownable2Step` from OpenZeppelin as opposed to `Ownable` as it gives you the security of not unintentionally sending the `owner` role to an address you do not control. Also, consider using only `onlyOwner` modifier instead of using both `onlyOwner` and `restricted` modifiers because they are basically the same, and using both only creates confusion.

[L-02] Discrepancy between comment and code

Function `manualsend` has the following comment:

```
* @dev Function to send all ETH balance of the contract to the controller
wallet. Only callable by the owner.
```

It says that the function is only callable by the owner. But there is not `onlyOwner` modifier neither some kind of check that the caller is the owner. Put the `onlyOwner` modifier in place or delete the comment.

The same applies for the `airdrop` function which can be called by anyone but the comment says it should be callable only by the owner.

[I-01] Using `SafeMath` when compiler is ^0.8.0

There is no need to use `SafeMath` when compiler is ^0.8.0 because it has built-in under/overflow checks. Also, you are both using methods from `SafeMath` and the normal arithmetic operators such as `*`, `/`, etc. Use them instead of `SafeMath` functions.

[I-02] Redundant modifier

In `manualswap()` we have two checks for the same thing - if the `msg.sender` is a particular address. Owner's and `controllerWallet`'s address is the same. So either remove the modifier `onlyOwner` or remove the require check:

```
function manualswap(uint256 amount) external onlyOwner {
 require(_msgSender() == controllerWallet);
...
```

[I-03] Unused code

`deadAddress` variable is not used anywhere in the contract.

```
address public constant deadAddress = address(0xdead);
```

as well as `event SwapAndLiquify()`:

```
event SwapAndLiquify(
        uint256 tokensSwapped,
        uint256 ethReceived,
        uint256 tokensIntoLiquidity
    );
```

Remove them if they won't be used.

[I-04] Prefer Solidity Custom Errors over `require` statements with strings

Using Solidity Custom Errors has the benefits of less gas spent in reverted transactions, better interoperability of the protocol as clients of it can catch the errors easily on-chain, as well as you can give descriptive names of the errors without having a bigger bytecode or transaction gas spending, which will result in a better UX as well. Consider replacing the `require` statements with custom errors.

[I-05] Missing event emissions in state changing methods

It's a best practice to emit events on every state changing method for off-chain monitoring.The following methods are missing event emissions, which should be added:

- `removeLimits()`
- `updateSwapTokensAtAmount()`
- `whitelistContract()`
- `verifyUser()`
- `excludeFromMaxTransaction()`
- `updateSwapEnabled()`

[I-06] Use a stable pragma statement

Using a floating pragma `^0.8.9` statement is discouraged as code can compile to different bytecodes with different compiler versions. Use a stable pragma statement to get a deterministic bytecode. Consider using a stable 0.8.19 version to make sure it is up to date.