HashEx
BLOCKCHAIN SECURITY

Messier

smart contracts final audit report

August 2023

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Messier team to perform an audit of their smart contract. The audit was conducted between 01/08/2023 and 04/08/2023.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code was provided directly in .sol file. The SHA-1 hash of audited contracts:

MESSIER M87.sol `a87ae5f93be41e8b6e4ac2a9c0a47e82a62bde1e`

Update. A recheck was done for a file with `24981dcfe55ffdf54b4f87311bf8696dbb6796ac` SHA-1 hash. This version was deployed to the Ethereum network at address 0x80122c6a83C8202Ea365233363d3f4837D13e888.

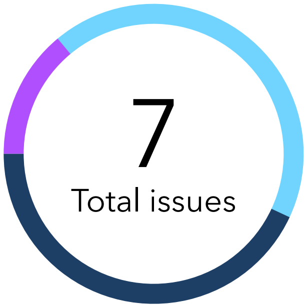
2.1 Summary

Project name	Messier
URL	https://messier.app
Platform	Ethereum
Language	Solidity

2.2 Contracts

Name	Address
M87	0x80122c6a83C8202Ea365233363d3f4837D13e888

3. Found issues



●	Medium	1 (14%)
●	Low	3 (43%)
●	Info	3 (43%)

C1. M87

ID	Severity	Title	Status
C1-01	● Medium	Indistinguishable fees	✓ Resolved
C1-02	● Low	Gas optimizations	✓ Partially fixed
C1-03	● Low	Adding liquidity	✓ Resolved
C1-04	● Low	Absence of event emission in setSwapEnabled() function	✓ Resolved
C1-05	● Info	Swaps and liquidity tokens	✓ Acknowledged
C1-06	● Info	Typos	✓ Resolved
C1-07	● Info	Usage of hardcoded addresses	✓ Resolved

4. Contracts

C1. M87

Overview

An ERC20 standard token contract. The initial supply is fixed, i.e. no minting functionality. The Messier M87 token supports fees on transfers to and from UniswapV2-like trading pair, the fees percent is limited up to 3% after the initial sale period of 30 minutes. In first 30 minutes of sale fees are increased up to 99% for the first block to protect users from bots.

Issues

C1-01 Indistinguishable fees ● Medium ✓ Resolved

The fees have different values depending on the trade direction and are split into marketing and liquidity fractions. In general, these fractions are different for buy and sell transfers, but the actual amounts aren't stored on-chain and are calculated according to the sum of fees:

```
uint256 liquidityFeeOnBuy;
uint256 liquidityFeeOnSell;

uint256 marketingFeeOnBuy;
uint256 marketingFeeOnSell;

uint256 _totalFeesOnBuy;
uint256 _totalFeesOnSell;

function _transfer(...) {
  ...
  uint256 contractTokenBalance = balanceOf(address(this));
  uint256 totalFee = _totalFeesOnBuy + _totalFeesOnSell;
  uint256 liquidityShare = liquidityFeeOnBuy + liquidityFeeOnSell;
  uint256 marketingShare = marketingFeeOnBuy + marketingFeeOnSell;

  uint256 liquidityTokens = contractTokenBalance * liquidityShare / totalFee;
```

```
    uint256 marketingTokens = contractTokenBalance * marketingShare / totalFee;
    ...
  }
```

For example, liquidityFeeOnBuy = 0%, marketingFeeOnBuy = 1%, liquidityFeeOnSell = 2%, marketingFeeOnSell = 3%, and there are 2 transactions: buy for 100 tokens and sell for 1000 tokens. The total collected fees are 1 + 50 = 51 tokens, which should be split as 20 for liquidity and 31 for marketing, but according to the code above liquidityTokens = 17 and marketingTokens = 34.

Recommendation

Fix the fees splitting by storing the amounts on-chain or add documentation for the current behavior.

Update

The swap for liquidity functionality was removed in the update.

C1-02 Gas optimizations ● Low ✓ Partially fixed

1. The variables `uniswapV2Router`, `uniswapV2Pair`, and `maxFee` should be declared as immutable.
2. Unnecessary reads from storage may be avoided by using local variables: in the `updateBuyFees(): liquidityFeeOnBuy, marketingFeeOnBuy, _totalFeesOnBuy`; in the `updateSellFees(): liquidityFeeOnSell, marketingFeeOnSell, _totalFeesOnSell`; in the `changeMarketingWallet(): marketingWallet`; in the `_transfer(): _totalFeesOnBuy, _totalFeesOnSell, liquidityFeeOnBuy, liquidityFeeOnSell, marketingFeeOnBuy, marketingFeeOnSell, tradingTime`; in the `setSwapTokensAtAmount(): swapTokensAtAmount`; in the `setEnableMaxTransactionLimit(): maxTransactionLimitEnabled`; in the `setMaxTransactionAmounts(): maxTransactionAmountBuy, maxTransactionAmountSell`.
3. Two separate swaps are performed in the `_transfer()` function. They should be merged into a single operation.

C1-03 Adding liquidity ● Low ✓ Resolved

Manual liquidity adding may occasionally fail due to re-entrancy to the Pair contract, i.e. `Pair.mint() -> Token.transfer() -> Token.swapAndLiquify() -> Pair.mint() -> revert("UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED")`. This action is triggered when the `swapAndLiquify()` function is called during the user's tokens being transferred to the pair contract. In that case these tokens would be consumed by adding liquidity inside `swapAndLiquify()` and mint for user would fail.

Recommendation

Users should trigger call of `swapAndLiquify()` manually by selling tokens to the pair or wait for someone else to sell.

Update

The swap for liquidity functionality was removed in the update.

C1-04 Absence of event emission in setSwapEnabled() function ● Low ✓ Resolved

In the smart contract, the function `setSwapEnabled()` changes the state of the `swapEnabled` variable but does not emit an event upon this value alteration. This omission obstructs the tracking and verification of `swapEnabled` state changes, limiting transparency and audibility.

C1-05 Swaps and liquidity tokens ● Info ✓ Acknowledged

The `swapAndLiquify()` and `swapAndSendMarketing()` perform swaps with 100% slippage, making it vulnerable to sandwich trades if fees are set to near zero.

The resulting liquidity tokens are sent to the `0xdead` address, denying possibility of liquidity migration.

C1-06 Typos ● Info ✓ Resolved

Typos reduce code readability. Typos in 'limis'.

C1-07 Usage of hardcoded addresses ● Info ✓ Resolved

The constructor uses hardcoded addresses for router, marketing wallet, pinklock. Usage of hardcoded addresses implicates testing. It should be noted that the address for router is set depending on the network id, but the pinklock address is set for all networks and will have the correct value only if the code is deployed on BSC network as on the other networks this address is EOA.

```
  constructor () ERC20("Messier", "M87")
  {
    address router;
    if (block.chainid == 56) {
      router = 0x10ED43C718714eb63d5aA57B78B54704E256024E; // BSC Pancake Mainnet
Router
    } else if (block.chainid == 97) {
      router = 0xD99D1c33F9fC3444f8101754aBC46c52416550D1; // BSC Pancake Testnet
Router
    } else if (block.chainid == 1 || block.chainid == 5) {
      router = 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D; // ETH Uniswap Mainnet %
Testnet
    } else {
      revert();
    }
    ...
    _isExcludedFromMaxTxLimit[0x407993575c91ce7643a4d4cCACc9A98c36eE1BBE] = true; //
pinklock
    ...
    _isExcludedFromMaxWalletLimit[0x407993575c91ce7643a4d4cCACc9A98c36eE1BBE] =
true; //pinklock
    ...
    _isExcludedFromFees[0x407993575c91ce7643a4d4cCACc9A98c36eE1BBE] = true; //pinklock
    ...
}
```

Update

Hardcoded addresses for pinklock were removed in the updated code.

5. Conclusion

1 medium, 3 low severity issues were found during the audit. 1 medium, 2 low issues were resolved in the update.

The reviewed contract is highly dependent on the owner’s account. Users using the project have to trust the owner and that the owner's account is properly secured.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

✉ contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx
BLOCKCHAIN SECURITY