SECURITY AUDIT OF

TROY MEME

Public Report

Dec 11, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Dec 11, 2024. We would like to thank the TROY for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Troy Meme. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Troy Meme

Troy Pump is a meme coin project developed by the meme coin and **Troy Dao** ecosystem lovers. **TroyPump.fun** is developed by the meme loving community and not associated with **Troy Dao Foundation**.

The project aims to create value and build a large meme coin community on the Troy Dao ecosystem.

1.2. Audit Scope

This audit focused on identifying security flaws in code and the design of the Troy Meme.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
3fd412dc95fcd71052e56c762323ce6ac6bd4b39acc36e748c2773a88b0bd9c3	./MemeManager.sol
7bafa37915d2a77e7ed459720e7410c14944a2e46d0daba90bf6652dea9dde3d	./MemeFactory.sol
1dba4e34f10117a237e2591cb7a6dbdfafd9ea4df8f5ae927c4e9d8fd068662a	./MemeToken.sol
c36bcb55d4900373cd4d0cded04854a2405030c69e022c3919626bde295fd60b	./libraries/LibCalculatePair.sol
1099a085afe5d7c06eba204a3ae6ae4f8a8792415e5fb94c6c29ace496d8bb53	./libraries/DataTypes.sol
99b81f076f57fd3017926602d9e00aa8db76c4a6dc7d5fd411970b9ea58209b6	./libraries/Math.sol
9647eb397b5bb8aea53876482e41a10f5459d53e9dd7d192bf2b35e9167e61ed	./libraries/Strings.sol
eae3511d540d5a6329b59d2fc8b92966b5bbe451d38c5938f10e16f3b6730605	./interfaces/IMemeToken.sol
0953150dfca6f3b8e8eb65f69b7bc8e3dec3caeb864961263bc70ef5fa0689ce	./interfaces/IMemeFactory.sol
17c3f40c3b9608ffd1a00f6d268a13b1c281fb5f0e5ab5e212d769d646c9a8b0	./interfaces/IWETH9.sol

c94588efb4f8220a278591d53a3857ed66dcba6d69858f473e9c1b3289e30f6a	./interfaces/IMemeManager.sol
8a0b91ca427e38e69f4c62460ed19c1d12ebbf29735b69a5ff6a06ee499dd8e4	./interfaces/INonfungiblePositionManager.sol

1.3. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.

SEVERITY LEVEL	DESCRIPTION
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

TROY acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. TROY understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, TROY agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the TROY will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the TROY, the final report will be considered fully accepted by the TROY without the signature.

2. AUDIT RESULT

2.1. Overview

The Troy Meme was written in `Solidity` language, with the required version to be `0.8.26` and built upon the **OpenZeppelin** library.

Note: The scope of the audit is limited to the source code files provided. `MemeFactory` and `MemeManager` contracts in the scope are **upgradeable** contracts, the contract owner can change the contract logic at any time in the future.

`MemeToken` is a custom **ERC20 Token** with a checking mechanism for `_enableLPTrading`. Before enabling, only transactions from or to the factory and manager are accepted. In addition, the factory contract can set the **Image URI**.

The `MemeFactory` contract is the foundation for creating and managing memes within the ecosystem. It is responsible for handling the lifecycle of memes, from their creation to the time they are added to the **Uniswap V3 Pool**. Each meme is assigned a unique **ID** and stored in the contract's state. The contract also includes functions calculating the value of **ETH** or selling tokens for users. In addition, this function supports selling Bonding and buying Bonding from **ETH** or other buying with Sharing amounts in user input.

The `MemeManager` contract does the step of adding the liquidity to **Uniswap V3 Pool** for **Meme Tokens** through `addLiquidityForMeme`. After adding the liquidity, if there is any amount of **Meme** or **ETH** still in the contract, the amount is refunded to the creator of the **Meme Tokens**. Only factory contract can call `addLiquidityForMeme`. The contract also supports the `collect` function for collecting the reward of **Uniswap V3** for the creator of **Meme Tokens**.

2.2. Findings

#	Title	Severity	Status
1	Rounding Error in `getBuyShareByEthValue`	HIGH	Fixed
2	ETH Refund Lockup Issue	MEDIUM	Fixed
3	Wrong value for constant `K`	INFORMATIVE	Acknowledged
4	Wrong parameter in require statement.	INFORMATIVE	Fixed
5	Typographical Error	INFORMATIVE	Partial-fixed
6	Leftover Console Log Statements	INFORMATIVE	Fixed

#	Title	Severity	Status
7	Wrong default value for `_protocolPercentage`	INFORMATIVE	Fixed

2.2.1. Rounding Error in `getBuyShareByEthValue`. HIGH

Position

- `MemeFactory.sol`#L209,210

Description

```
function getBuyShareByEthValue(
        uint256 supply,
        uint256 selledEthValue,
        uint256 ethBuyValue
    ) public pure returns (uint256) {
        // y = 1073000191 - 32190005730/(30+x)
        // y = (1073000191 * 10^18) - ((32190005730*(10^18)/((30*10^18)+x)) * (10^18))
        uint256 X = X_0 + (selledEthValue + ethBuyValue);
        uint256 KDivX = K / X;
        uint256 new_y = Y_0 - (KDivX * 10 ** DECIMALS);
        uint256 summation = new_y - supply;

        return summation;
    }
```

In the function `getBuyShareByEthValue`, there is a precision loss caused by performing division `(K / X)` before multiplication. Since all numbers are integer arithmetic, any division operation will cause floor rounding. When this truncated result is multiplied, leading to inaccurate results. The value of the subtraction `Y_0 - result` ends up being significantly larger. Users can earn much more token since `new_y` is larger than expected.

```
uint256 KDivX = K / X;
    uint256 new_y = Y_0 - (KDivX * 10 ** DECIMALS);
```

RECOMMENDATION

The multiplication should be calculated before division.

UPDATES

- *Dec 9, 2024*: The team has acknowledged the issue and has implemented the recommended changes.

2.2.2. ETH Refund Lockup Issue. MEDIUM

Position

- `MemeManager.sol`#addLiquidityForMeme()

Description

```
function addLiquidityForMeme(
        address memeAddr,
        uint256 tokenAmount,
        uint256 ethVaule
    ) public override onlyFactory returns (bool) {

    ...

        uint256 leftToken = IMemeToken(memeAddr).balanceOf(address(this));
        address creator = IMemeToken(memeAddr).owner();
        if (leftToken > 0) {
            IMemeToken(memeAddr).transfer(creator, leftToken);
        }

        INonfungiblePositionManager(v3NonfungiblePositionManagerAddress)
            .refundETH();
        // Keep refundETH on manager contract
        // if (address(this).balance > 0) {
        //     (bool success, ) = payable(creator).call{
        //         value: address(this).balance
        //     }("");
        //     if (!success) {
        //         revert SendETHFailed();
        //     }
        // }
        return true;
    }
```

As the mechanism, after minting liquidity on **Uniswap V3**, there might be remaining **MemeToken** or **ETH** amount. The remaining amount will be sent back to the creator of **MemeToken**.

```
/// @inheritdoc IPeripheryPayments
    function refundETH() external payable override {
        if (address(this).balance > 0) TransferHelper.safeTransferETH(msg.sender,
address(this).balance);
    }
```

The `refundETH` method of the **Uniswap V3** `NonfungiblePositionManager` only returns excess **ETH** to the manager contract (`msg.sender`). This refunded **ETH** is locked within the manager contract, as there is no mechanism in place to transfer the excess **ETH** back to the creator.

Additionally, the factory contract does not include an implementation for tracking the `selledEthValue` after invoking the `addLiquidityForMeme` function.

Furthermore, the commented-out code designed to transfer the excess **ETH** back to the creator using a call operation is currently inactive.

UPDATES

- *Dec 9, 2024*: The team has acknowledged the issue and has implemented the changes. The team has added checking mechanism in new function `_addLiquidity`. But the `refundETH` of the **Uniswap V3** `NonfungiblePositionManager` only returns excess **ETH** to the contract (`msg.sender`). So this mechanism should be implemented in **`MemeManager.sol`#addLiquidityForMeme()**
- *Dec 10, 2024*: The team has acknowledged the issue and has moved the mechanism to `MemeManager.sol`.

2.2.3. Wrong value for constant `K`. INFORMATIVE

Position

- `MemeFactory.sol`#23,24,25

Description

```
...
    uint256 constant K = 8047501432 * (10 ** DECIMALS);
    uint256 constant X_0 = 75 * (10 ** (DECIMALS - 1));
    uint256 constant Y_0 = 1073000191 * (10 ** DECIMALS);
...
```

This factory uses **Constant-product** invariant `x * y = k` for calculating prices. The constant `K` is defined by `X_0 * Y_0` but the correct value should has been `80475014325 * (10 ** (DECIMALS - 1))` instead of `8047501432 * (10 ** DECIMALS)`. This led to a loss of value when calculating prices.

RECOMMENDATION

Checking the correct value of constant `K`.

UPDATES

- *Dec 9, 2024*: The team has acknowledged the information.

2.2.4. Wrong parameter in require statement. INFORMATIVE

Position

- `MemeFactory.sol`#setSqrtPriceX96()

Description

According to the logic of the function, the condition that needs to be checked is `sqrtPriceX96_` instead of `sqrtPriceX96`. This parameter might have been confused.

```
function setSqrtPriceX96(uint160 sqrtPriceX96_) public onlyOwner {
        require(sqrtPriceX96 > 0, "sqrtPriceX96 must be greater than 0!");

        sqrtPriceX96 = sqrtPriceX96_;
        emit SetSqrtPriceX96(sqrtPriceX96_);
    }
```

UPDATES

- *Dec 9, 2024*: The team has acknowledged the issue and has implemented the changes.

2.2.5. Typographical Error. INFORMATIVE

There are still some instances of spelling errors in the variables and comments within the code such as `selledEthValue` and `buyShahre` in factory, `tansfer` in token,... These typos could make the code harder to read and understand.

UPDATES

- *Dec 9, 2024*: The team has acknowledged and fixed some typographical errors, but there are still some spelling errors that have been overlooked such as `selledEthValue`,...

2.2.6. Leftover Console Log Statements INFORMATIVE

Position

- `MemeFactory.sol`#L507,584

Description

There are still some instances of `console.log` statements in the code that are unnecessary. These logs may be leftover from debugging and do not serve any purpose in the final implementation. It is encouragement to review and remove these logs to ensure cleaner and more efficient code.

UPDATES

- *Dec 9, 2024*: The team has acknowledged and commented out `console.log`.

2.2.7. Wrong default value for `_protocolPercentage` INFORMATIVE

Position

- `MemeManager.sol`#L55,110

Description

There is an inconsistency between the documentation and the implementation regarding the default liquidity reward fee. In the identifier comment, the default fee is stated as 10%, but in the initialize function, `_protocolPercentage` is set to 5000, which corresponds to 50% (assuming the value is in basis points).

RECOMMENDATION

The default `_protocolPercentage` value should be checked.

UPDATES

- *Dec 11, 2024*: The team has acknowledged and changed `_protocolPercentage` value to `1000`.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Dec 9, 2024*	Private Report	Verichains Lab
1.1	*Dec 10, 2024*	Private Report	Verichains Lab
1.2	*Dec 11, 2024*	Public Report	Verichains Lab

Table 2. Report versions history