

SMART CONTRACT

Security Audit Report

Project: Memecoin
Website: memecoin.org
Platform: Ethereum
Language: Solidity
Date: May 11th, 2024

Table of contents

Introduction	4
Project Background	4
Audit Scope	5
Claimed Smart Contract Features	6
Audit Summary	7
Technical Quick Stats	8
Business Risk Analysis	9
Code Quality	10
Documentation	10
Use of Dependencies	10
AS-IS overview	11
Severity Definitions	12
Audit Findings	13
Conclusion	16
Our Methodology	17
Disclaimers	19
Appendix	
• Code Flow Diagram	20
• Slither Results Log	21
• Solidity static analysis	23
• Solhint Linter	25

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

Introduction

As part of EtherAuthority's community smart contracts audit initiatives, the smart contracts of Memecoin from memecoin.org were audited. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on May 11th, 2024.

The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contract.

Project Background

- The Memecoin contract is designed to be a secure and functional ERC20 token with:
 - Standard ERC20 token capabilities.
 - Support for gasless approvals using the ERC20Permit extension.
 - Ownership control through the Ownable contract.
 - Restriction on permit function to ensure only a specific token pool can be approved for spending.
- This design ensures that only authorized interactions can occur with the token via the permit function, enhancing security. Additionally, the owner has the flexibility to set and update the token pool as needed.
- Memecoin contracts offer various functions such as renouncement, transfer of ownership, permit, and setting the new TokenPool address.
- Overall, this contract implements a token with permit functionality, where the spender is restricted to the token pool address. Additionally, it provides ownership functionality and allows the owner to update the token pool address.

Audit scope

Name	Code Review and Security Analysis Report for Memecoin Smart Contract
Platform	Ethereum
Language	Solidity
File	Memecoin.sol
Ethereum Code	0xb131f4a55907b10d1f0a50d8ab8fa09ec342cd74
Smart Contract Date	May 11th, 2024

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
<p>Tokenomics:</p> <ul style="list-style-type: none">• Name: Memecoin• Symbol: MEME• Decimals: 18• Total Supply: 69 billion	<p>YES, This is valid.</p>
<p>Ownership control:</p> <ul style="list-style-type: none">• The new TokenPool address can be set by the owner.• The current owner can transfer the ownership.• The owner can renounce ownership.	<p>YES, This is valid.</p> <p>We suggest renouncing ownership once the ownership functions are not needed. This is to make the smart contract 100% decentralized.</p>

Audit Summary

According to the standard audit assessment, the Customer's solidity-based smart contracts are "**Secured**". Also, these contracts contain owner control, which does not make them fully decentralized.



We used various tools like Slither, Solhint, and Remix IDE. At the same time, this finding is based on a critical analysis of the manual audit.

All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit Overview section. The general overview is presented in the AS-IS section and all identified issues can be found in the Audit overview section.

We found 0 critical, 0 high, 0 medium, 0 low, and 3 very low level issues.

Investor Advice: A technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner-controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

Technical Quick Stats

Main Category	Subcategory	Result
Contract Programming	The solidity version is not specified	Passed
	Solidity version is too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Features claimed	Passed
	Other programming issues	Moderated
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Passed
Gas Optimization	"Out of Gas" Issue	Passed
	High consumption 'for/while' loop	Passed
	High consumption 'storage' storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage is not set	Passed
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

Overall Audit Result: PASSED

Business Risk Analysis

Category	Result
● Buy Tax	0%
● Sell Tax	0%
● Cannot Buy	Not Detected
● Cannot Sell	Not Detected
● Max Tax	0%
● Modify Tax	Not Detected
● Fee Check	No
● Is Honeypot	Not Detected
● Trading Cooldown	Not Detected
● Can Pause Trade?	No
● Pause Transfer?	No
● Max Tax?	No
● Is it Anti-whale?	No
● Is Anti-bot?	Not Detected
● Is it a Blacklist?	Not Detected
● Blacklist Check	No
● Can Mint?	No
● Is it Proxy?	Not Detected
● Can Take Ownership?	Yes
● Hidden Owner?	Not Detected
● Self Destruction?	Not Detected
● Auditor Confidence	High

Overall Audit Result: PASSED

Code Quality

This audit scope has 1 smart contract. Smart contracts contain Libraries, Smart contracts, inherits, and Interfaces. This is a compact and well-written smart contract.

The libraries in Memecoin are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties/methods can be reused many times by other contracts in the Memecoin.

The EtherAuthority team has no scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Code parts are well commented on in the smart contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a Memecoin smart contract code in the form of an [Etherscan](#) web link.

As mentioned above, code parts are well commented on. and the logic is straightforward. So it is easy to quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Use of Dependencies

As per our observation, the libraries used in this smart contract infrastructure that are based on well-known industry standard open-source projects.

Apart from libraries, its functions are not used in external smart contract calls.

AS-IS overview

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	permit	write	Passed	No Issue
3	setTokenPool	external	Missing Zero Address Validation, Centralized risk	Refer Audit Findings
4	permit	write	Passed	No Issue
5	nonces	read	Passed	No Issue
6	DOMAIN_SEPARATOR	external	Passed	No Issue
7	_useNonce	internal	Passed	No Issue
8	name	read	Passed	No Issue
9	symbol	read	Passed	No Issue
10	decimals	read	Passed	No Issue
11	totalSupply	read	Passed	No Issue
12	balanceOf	read	Passed	No Issue
13	transfer	write	Passed	No Issue
14	allowance	read	Passed	No Issue
15	approve	write	Passed	No Issue
16	transferFrom	write	Passed	No Issue
17	increaseAllowance	write	Passed	No Issue
18	decreaseAllowance	write	Passed	No Issue
19	_transfer	internal	Passed	No Issue
20	_mint	internal	Passed	No Issue
21	burn	internal	Passed	No Issue
22	_approve	internal	Passed	No Issue
23	spendAllowance	internal	Passed	No Issue
24	_beforeTokenTransfer	internal	Passed	No Issue
25	_afterTokenTransfer	internal	Passed	No Issue
26	onlyOwner	modifier	Passed	No Issue
27	owner	read	Passed	No Issue
28	_checkOwner	internal	Passed	No Issue
29	renounceOwnership	write	Centralized risk	Refer Audit Findings
30	transferOwnership	write	Centralized risk	Refer Audit Findings
31	_transferOwnership	internal	Passed	No Issue
32	_domainSeparatorV4	internal	Passed	No Issue
33	buildDomainSeparator	read	Passed	No Issue
34	_hashTypedDataV4	internal	Passed	No Issue
35	eip712Domain	read	Passed	No Issue

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g. public access to crucial
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets, that can't have a significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations, and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

No Medium-severity vulnerabilities were found.

Low

No Low severity vulnerabilities were found.

Very Low / Informational / Best practices:

(1) Floating Pragma: [Memecoin.sol](#)

```
pragma solidity ^0.8.18;
```

This contract may not function as expected due to inconsistent solidity compiler versions being specified.

Resolution: Specify and lock the Solidity pragma version to a fixed and known version in your smart contract to avoid potential issues caused by unexpected compiler behavior in future versions.

(2) Missing Zero Address Validation: [Memecoin.sol](#)

```
function setTokenPool(address _tokenPool) external onlyOwner {  
    tokenPool = _tokenPool;  
    emit TokenPoolUpdated(_tokenPool);  
}
```

Some functions in this contract may not appropriately check for zero addresses being used.

Resolution: Implement zero address validation checks in relevant functions to prevent unintended behavior and enhance the contract's security.

(3) Centralized risk:

Memecoin.sol

```
function setTokenPool(address _tokenPool) external onlyOwner {
    tokenPool = _tokenPool;
    emit TokenPoolUpdated(_tokenPool);
}
```

The onlyOwner can call the setTokenPool function.

Ownable.sol

```
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}

function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _transferOwnership(newOwner);
}
```

The onlyOwner can call the renounceOwnership function and transferOwnership function.

Resolution: To make the smart contract 100% decentralized. We suggest renouncing ownership of the smart contract once its function is completed.

Centralization

This smart contract has some functions which can be executed by the Admin (Owner) only. If the admin wallet's private key would be compromised, then it would create trouble. The following are Admin functions:

Memecoin.sol

- setTokenPool: The new TokenPool address can be set by the owner.

Ownable.sol

- _checkOwner: Throws if the sender is not the owner.
- renounce Ownership: Deleting ownership will leave the contract without an owner, removing any owner-only functionality.
- transferOwnership: The current owner can transfer ownership of the contract to a new account.

To make the smart contract 100% decentralized, we suggest renouncing ownership of the smart contract once its function is completed.

Conclusion

We were given a contract code in the form of [Etherscan](#) web links. And we have used all possible tests based on given objects as files. We observed 3 Informational issues in the smart contracts. but those are not critical. So, **it's good to go for the production.**

Since possible test cases can be unlimited for such smart contracts protocol, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover the maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in the As-is overview section of the report.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

The security state of the reviewed smart contract, based on standard audit procedure scope, is **"Secured"**.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of the systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and white box penetration testing. We look at the project's website to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

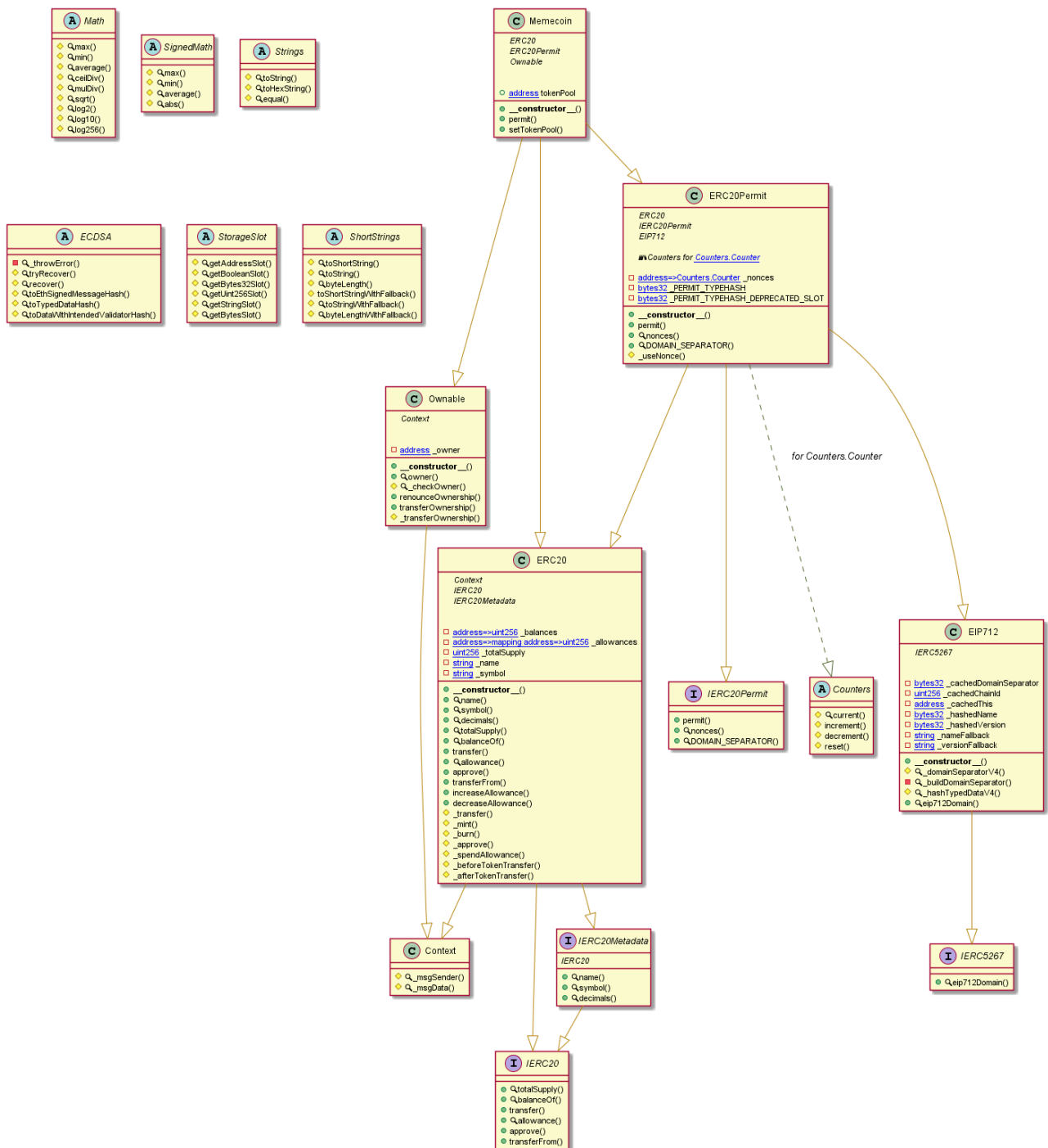
Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

Appendix

Code Flow Diagram - Memecoin



This is a private and confidential document. No part of this document should be disclosed to third party without prior written permission of EtherAuthority.

Email: audit@EtherAuthority.io

Slither Results Log

Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We did the analysis of the project altogether. Below are the results.

Slither Log >> Memecoin.sol

```
ERC20Permit.constructor(string).name (Memecoin.sol#849) shadows:
- ERC20.name() (Memecoin.sol#708-710) (function)
- IERC20Metadata.name() (Memecoin.sol#305) (function)
Memecoin.constructor(string,string,uint256,address).name (Memecoin.sol#894) shadows:
- ERC20.name() (Memecoin.sol#708-710) (function)
- IERC20Metadata.name() (Memecoin.sol#305) (function)
Memecoin.constructor(string,string,uint256,address).symbol (Memecoin.sol#894) shadows:
- ERC20.symbol() (Memecoin.sol#712-714) (function)
- IERC20Metadata.symbol() (Memecoin.sol#307) (function)
Memecoin.permit(address,address,uint256,uint256,uint8,bytes32,bytes32).owner (Memecoin.sol#902) shadows:
- Ownable.owner() (Memecoin.sol#668-670) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing

Memecoin.setTokenPool(address)._tokenPool (Memecoin.sol#910) lacks a zero-check on :
- tokenPool = _tokenPool (Memecoin.sol#911)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

ERC20Permit.permit(address,address,uint256,uint256,uint8,bytes32,bytes32) (Memecoin.sol#851-870) uses timestamp for comparisons
Dangerous comparisons:
- require(bool,string)(block.timestamp <= deadline,ERC20Permit: expired deadline) (Memecoin.sol#860)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Math.mulDiv(uint256,uint256,uint256) (Memecoin.sol#27-76) uses assembly
- INLINE ASM (Memecoin.sol#31-35)
- INLINE ASM (Memecoin.sol#45-50)
- INLINE ASM (Memecoin.sol#54-60)
Strings.toString(uint256) (Memecoin.sol#344-362) uses assembly
- INLINE ASM (Memecoin.sol#349-351)
- INLINE ASM (Memecoin.sol#354-356)
ECDSA.tryRecover(bytes32,bytes) (Memecoin.sol#416-430) uses assembly
- INLINE ASM (Memecoin.sol#421-425)
ECDSA.toEthSignedMessageHash(bytes32) (Memecoin.sol#469-475) uses assembly
- INLINE ASM (Memecoin.sol#470-474)
ECDSA.toTypedDataHash(bytes32,bytes32) (Memecoin.sol#481-489) uses assembly

ERC20Permit.constructor(string).name (Memecoin.sol#849) shadows:
- ERC20.name() (Memecoin.sol#708-710) (function)
- IERC20Metadata.name() (Memecoin.sol#305) (function)
Memecoin.constructor(string,string,uint256,address).name (Memecoin.sol#894) shadows:
- ERC20.name() (Memecoin.sol#708-710) (function)
- IERC20Metadata.name() (Memecoin.sol#305) (function)
Memecoin.constructor(string,string,uint256,address).symbol (Memecoin.sol#894) shadows:
- ERC20.symbol() (Memecoin.sol#712-714) (function)
- IERC20Metadata.symbol() (Memecoin.sol#307) (function)
Memecoin.permit(address,address,uint256,uint256,uint8,bytes32,bytes32).owner (Memecoin.sol#902) shadows:
- Ownable.owner() (Memecoin.sol#668-670) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing

Memecoin.setTokenPool(address)._tokenPool (Memecoin.sol#910) lacks a zero-check on :
- tokenPool = _tokenPool (Memecoin.sol#911)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

ERC20Permit.permit(address,address,uint256,uint256,uint8,bytes32,bytes32) (Memecoin.sol#851-870) uses timestamp for comparisons
Dangerous comparisons:
- require(bool,string)(block.timestamp <= deadline,ERC20Permit: expired deadline) (Memecoin.sol#860)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Math.mulDiv(uint256,uint256,uint256) (Memecoin.sol#27-76) uses assembly
- INLINE ASM (Memecoin.sol#31-35)
- INLINE ASM (Memecoin.sol#45-50)
- INLINE ASM (Memecoin.sol#54-60)
Strings.toString(uint256) (Memecoin.sol#344-362) uses assembly
- INLINE ASM (Memecoin.sol#349-351)
- INLINE ASM (Memecoin.sol#354-356)
ECDSA.tryRecover(bytes32,bytes) (Memecoin.sol#416-430) uses assembly
- INLINE ASM (Memecoin.sol#421-425)
ECDSA.toEthSignedMessageHash(bytes32) (Memecoin.sol#469-475) uses assembly
- INLINE ASM (Memecoin.sol#470-474)
ECDSA.toTypedDataHash(bytes32,bytes32) (Memecoin.sol#481-489) uses assembly
```

This is a private and confidential document. No part of this document should be disclosed to third party without prior written permission of EtherAuthority.

Email: audit@EtherAuthority.io

Solidity Static Analysis

Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

Memecoin.sol

Inline assembly:

The Contract uses inline assembly, this is only advised in rare cases. Additionally static analysis modules do not parse inline Assembly, this can lead to wrong analysis results.

[more](#)

Pos: 1012:15:

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.

[more](#)

Pos: 1704:23:

Gas costs:

Gas requirement of function Memecoin.permit is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)

Pos: 1761:11:

Similar variable names:

Memecoin.(string,string,uint256,address) : Variables have very similar names "_totalSupply" and "totalSupply_". Note: Modifiers are currently not considered by this static analysis.

Pos: 1756:31:

No return:

`StorageSlot.getBytesSlot(bytes)`: Defines a return type but never explicitly returns a value.

Pos: 946:11:

Guard conditions:

Use `"assert(x)"` if you never ever want `x` to be false, not in any circumstance (apart from a bug in your code). Use `"require(x)"` if `x` can be false, due to e.g. invalid input or a failing external component.

[more](#)

Pos: 1704:15:

Guard conditions:

Use `"assert(x)"` if you never ever want `x` to be false, not in any circumstance (apart from a bug in your code). Use `"require(x)"` if `x` can be false, due to e.g. invalid input or a failing external component.

[more](#)

Pos: 1711:15:

Data truncated:

Division of integer values yields an integer value again. That means e.g. $10 / 100 = 0$ instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.

Pos: 181:35:

Solhint Linter

Linters are the utility tools that analyze the given source code and report programming errors, bugs, and stylistic errors. For the Solidity language, there are some linter tools available that a developer can use to improve the quality of their Solidity contracts.

Memecoin.sol

```
Compiler version ^0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:3
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 13:61
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 13:84
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 13:98
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 13:580
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 17:586
Error message for revert is too long
Pos: 13:659
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 13:691
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:797
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:827
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:877
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:887
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:897
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:907
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:917
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:927
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:937
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:947
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:1011
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:1124
Explicitly mark visibility in function (Set ignoreConstructors to
```

```
true if using solidity >=0.7.0)
Pos: 5:1242
Error message for require is too long
Pos: 9:1284
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:1344
Error message for require is too long
Pos: 9:1490
Error message for require is too long
Pos: 9:1513
Error message for require is too long
Pos: 9:1514
Error message for require is too long
Pos: 9:1519
Error message for require is too long
Pos: 9:1568
Error message for require is too long
Pos: 9:1573
Error message for require is too long
Pos: 9:1599
Error message for require is too long
Pos: 9:1600
Code contains empty blocks
Pos: 94:1638
Code contains empty blocks
Pos: 93:1654
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:1689
Code contains empty blocks
Pos: 55:1689
Avoid making time-based decisions in your business logic
Pos: 17:1703
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:1751
```

Software analysis result:

This software reported many false positive results and some are informational issues. So, those issues can be safely ignored.



This is a private and confidential document. No part of this document should be disclosed to third party without prior written permission of EtherAuthority.

Email: audit@EtherAuthority.io