Audit Report

MEPE

October 2025

Files: Token.sol, MultisigManager.sol, ManagedToken.sol

Audited by AI Gemini

Table of Contents

Risk Classificatikon

The criticality of findings in AI Gemini smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation:** This considers how easily an attack can be executed, including the economic feasibility for an attacker.

2. **Impact of Exploitation:** This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical:** Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.

2. **Medium:** Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.

3. **Minor:** Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.

4. **Informative:** Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Overview

The MEPE project completed an audit of its core smart contracts, specifically focusing on T oken.sol, MultisigManager.sol, and ManagedT oken.sol. MEPE stands out as a promising initiative with a steadily growing community. The audit covered key aspects of the project's smart contracts including the token contract itself, its interface, and a multi-signature contract that enables authorised entities to execute administrative actions on the token.

Specifically, the multi-signature contract introduces a majority-based mechanism, where eligible participants can submit requests to perform administrative tasks on the token contract. These tasks may include minting new tokens, blacklisting addresses and burning their tokens, pausing the token contract, or modifying its ownership. A request is approved only when a majority of the designated voters consent to the action.

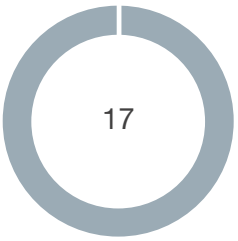
Through these contracts, the MEPE project showcases an innovative approach to managing token contracts by leveraging a multi-signature system for enhanced decentralisation.

Review

Audit Updates

Initial Audit	01 Oct 2025

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	17

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	17	0	0	0

Diagnostics

Severity	Code	Description	Status
●	AOI	Arithmetic Operations Inconsistency	Unresolved
●	BC	Blacklists Addresses	Unresolved
●	BBT	Burns Blacklisted Tokens	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	IDI	Immutable Declaration	Unresolved
●	IRE	Indefinite Request Expiry	Unresolved
●	MT	Mints Tokens Improvement	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MU	Modifiers Usage	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RSRS	Redundant SafeMath Require Statement	Unresolved
●	ST	Stops Transactions	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

•	L23	ERC20 Interface Misuse	Unresolved

AOI - Arithmetic Operations Inconsistency

Criticality	Minor / Informative
Location	Token.sol#L702
Status	Unresolved

Description

The contract uses both the SafeMath library and native arithmetic operations. The SafeMath library is commonly used to mitigate vulnerabilities related to integer overflow and underflow issues. However, it was observed that the contract also employs native arithmetic operators (such as +, -, *, /) in certain sections of the code.

The combination of SafeMath library and native arithmetic operations can introduce inconsistencies and undermine the intended safety measures. This discrepancy creates an inconsistency in the contract's arithmetic operations, increasing the risk of unintended consequences such as inconsistency in error handling, or unexpected behavior.

```
function issue(
        uint amount,
        address to
    ) external onlyOwner whenNotDeprecated {
        require(_totalSupply + amount > _totalSupply);
        require(balances[to] + amount > balances[to]);

        balances[to] = balances[to].add(amount);
        _totalSupply = _totalSupply.add(amount);
        emit Issue(amount, to);
}
```

Recommendation

To address this finding and ensure consistency in arithmetic operations, it is recommended to standardize the usage of arithmetic operations throughout the contract. The contract should be modified to either exclusively use SafeMath library functions or entirely rely on native arithmetic operations, depending on the specific requirements and design

considerations. This consistency will help maintain the contract's integrity and mitigate potential vulnerabilities arising from inconsistent arithmetic operations.

BC - Blacklists Addresse

Criticality	Minor / Informative
Location	Token.sol#L313
Status	Unresolved

Description

The contract owner has the authority to stop addresses from transactions. The owner may take advantage of it by calling the `addBlackList()` function.

```
function addBlackList(address _evilUser) external onlyOwner {
        isBlackListed[_evilUser] = true;
        emit AddedBlackList(_evilUser);
}
```

Recommendation

The team should carefully manage the private keys of the owner's account.In the current context, the team should ensure that the ownership of the Token contract is securely assigned to the `MultisigManager.sol` contract at the time of deployment. This setup ensures that all critical functions are protected by the multi-signature mechanism, thereby reducing the risk of a single point of failure or unauthorized access.

Additionally, it is advised to validate that the deployment and initialization process of the Token contract does not inadvertently allow the contract to be owned by any account other than the intended `MultisigManager.sol` contract.

Suggested Solutions:

These measures, while improving security, do not eliminate the severity of the finding:

- Ensure that the signers of the multi-signature contract are distinct, independent entities with rigorously managed private keys.
- Ensure the multi-signature contract `MultisigManager.sol` is properly configured and operational from deployment.

- Ensure that the ownership of the Token contract is irrevocably assigned to the `MultisigManager.sol` contract and cannot be modified using the `transferOwnership()` method.

AOI - Arithmetic Operations Inconsistency

Criticality	Minor / Informative
Location	Token.sol#L329
Status	Unresolved

Description

The contract owner has the authority to burn tokens from a blacklisted address. The owner may take advantage of it by calling the `destroyBlackFunds()` function. As a result, the targeted address will lose the corresponding tokens.

```
function destroyBlackFunds(address _blackListedUser) external
onlyOwner {
    require(isBlackListed[_blackListedUser], "account not
blacklisted");
    uint dirtyFunds = balanceOf(_blackListedUser);
    balances[_blackListedUser] = 0;
    _totalSupply -= dirtyFunds;
    emit DestroyedBlackFunds(_blackListedUser, dirtyFunds);
)
```

Recommendation

The team should carefully manage the private keys of the owner's account.In the current context, the team should ensure that the ownership of the T oken contract is securely assigned to the `MultisigManager.sol` contract at the time of deployment. This setup ensures that all critical functions are protected by the multi-signature mechanism, thereby reducing the risk of a single point of failure or unauthorized access.

Additionally, it is advised to validate that the deployment and initialization process of the Token contract does not inadvertently allow the contract to be owned by any account other than the intended `MultisigManager.sol` contract.

Suggested Solutions:

These measures, while improving security, do not eliminate the severity of the finding:

- Ensure that the signers of the multi-signature contract are distinct, independent entities with rigorously managed private keys.
- Ensure the multi-signature contract `MultisigManager.sol` is properly configured and operational from deployment.
- Ensure that the ownership of the Token contract is irrevocably assigned to the `MultisigManager.sol` contract and cannot be modified using the `transferOwnership()` method.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Token.sol#L278,286,313,321,329,409,702,717
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function pause() external onlyOwner whenNotPaused {
        ...
}
function unpause() external onlyOwner whenPaused {
        ...
}
function addBlackList(address _evilUser) external onlyOwner {
        ...
}
function removeBlackList(address _clearedUser) external
onlyOwner {
        ...
}
function destroyBlackFunds(address _blackListedUser) external
onlyOwner {
        ...
}
function deprecate(address _upgradedAddress) external onlyOwner
{
        ...
}
function issue(uint amount,address to) external onlyOwner
        whenNotDeprecated {
        ...
}
function redeem(uint amount) external onlyOwner
whenNotDeprecated {
        ...}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself.In addition, ownership should be permanently assigned to the `MultisigManager.sol` contract, with independent signers whose interests are distinct and private keys securely managed. Consider disabling ownership transfers and exploring further decentralization mechanisms to minimize reliance on a single entity.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	Token.sol#L516 MultisigManager.sol#L161
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
decimals
votingAccountsNumber
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

IRE - Indefinite Request Expiry

Criticality	Minor / Informative
Location	MultisigManager.sol#L97
Status	Unresolved

Description

The `MultisigManager.sol` contract manages the creation and approval of administrative actions for the `Token.sol` contract. However, requests can remain pending indefinitely, with no requirement for voters to participate or to submit their votes within a set timeframe. This allows voters to delay their decisions or activate past proposals when conditions are more favourable to their interests. This introduces a centralization risk, as a small number of individuals could control or influence the timing and execution of critical actions.

```
struct Request {
        mapping(address => bool) approvedBy;
        uint approvals;
        bool completed;
        /** @dev `votingAccountsListGeneration` at moment of
this request creation */
        uint generation;
}
```

```
function _makeRequest() private returns (bytes32 reqId) {
    require(isVotingAccount[msg.sender], "not a voting
account");
    reqId = keccak256(
            abi.encode(
                requestCount++,
                blockhash(block.number - 1)
            )
        );
    assert(requests[reqId].approvals == 0); // Check for
request id collision
    requests[reqId].approvedBy[msg.sender] = true;
    requests[reqId].approvals = 1;
    requests[reqId].generation = votingAccountsListGeneration;
}
```

Recommendation

The team is advised to implement a mechanism that enforces a time limit for request validity. Requests should automatically expire if not approved within the specified timeframe, preventing the activation of outdated proposals and reducing centralization risk.

MT - Mints Tokens

Criticality	Minor / Informative
Location	Token.sol#L702
Status	Unresolved

Description

The contract owner has the authority to mint tokens. The owner may take advantage of it by calling the `issue()` function. As a result, the contract tokens will be highly inflated.

```
function issue(
    uint amount,
    address to
    ) external onlyOwner whenNotDeprecated {
    require(_totalSupply + amount > _totalSupply);
    require(balances[to] + amount > balances[to]);

    balances[to] = balances[to].add(amount);
    _totalSupply = _totalSupply.add(amount);
    emit Issue(amount, to);
}
```

Recommendation

The team should carefully manage the private keys of the owner's account.In the current context, the team should ensure that the ownership of the Token contract is securely assigned to the `MultisigManager.sol` contract at the time of deployment. This setup ensures that all critical functions are protected by the multi-signature mechanism, thereby reducing the risk of a single point of failure or unauthorized access.
MultisigManager.sol Additionally, it is advised to validate that the deployment and initialization process of the
T oken contract does not inadvertently allow the contract to be owned by any account other than the intended `MultisigManager.sol` contract.

Suggested Solutions:

These measures, while improving security, do not eliminate the severity of the finding:

- Ensure that the signers of the multi-signature contract are distinct, independent entities with rigorously managed private keys.
- Ensure the multi-signature contract `MultisigManager.sol` is properly configured and operational from deployment.
- Ensure that the ownership of the T oken contract is irrevocably assigned to the `MultisigManager.sol` contract and cannot be modified using the `transferOwnership()` method.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	Token.sol#L194,230,235,237,238,420,706,707 MultisigManager.sol#L259,319,361,415,468,521,583,584,647,648,709,710
Status	Unresolved

Description

he contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(_value == 0 || allowed[msg.sender][_spender] == 0)
require(_tos.length == _values.length)
require(to != address(0))
require(amount > 0)
require(amount <= senderBalance)
require(upgradedAddress == msg.sender)
require(_totalSupply + amount > _totalSupply)
require(balances[to] + amount > balances[to])
require(addVoters.length > 0 || removeVoters.length > 0)
require(address(token) != address(0))
require(upgraded != address(0))
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Token.sol#L124
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
constructor(uint _initialSupply, address _supplier) {
    ...
    balances[_supplier] = _initialSupply;
}
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	MultisigManager.sol#L319,361,415,468,521,583,647,709
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function. current state of the contract.

```
require(address(token) != address(0));
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	Token.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to v0.8.30+ the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily in cases where the explanatory error message is not used.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. Since the version of the contract is greater than `v0.8.30+` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

RSRS - Redundant SafeMath Require Statement

Criticality	Minor / Informative
Location	Token.sol#L32
Status	Unresolved

Description

The contract utilizes a `require` statement within the `add` function aiming to prevent overflow errors. This function is designed based on the SafeMath library's principles. In Solidity version 0.8.0 and later, arithmetic operations revert on overflow and underflow, making the overflow check within the function redundant. This redundancy could lead to extra gas costs and increased complexity without providing additional security.

```
function add(uint256 a, uint256 b) internal pure returns
(uint256) {
        uint256 c = a + b;
        require(c >= a);
        return c;
}
```

Recommendation

It is recommended to remove the `require` statement from the add function since the contract is using a Solidity pragma version equal to or greater than 0.8.0. By doing so, the contract will leverage the built-in overflow and underflow checks provided by the Solidity language itself, simplifying the code and reducing gas consumption. This change will uphold the contract's integrity in handling arithmetic operations while optimizing for efficiency and cost-effectiveness.

ST - Stops Transactions

Criticality	Minor / Informative
Location	Token.sol#L278
Status	Unresolved

Description

The contract owner has the authority to stop transactions for all users. The owner may take advantage of it by calling the `pause()` function.

```
function pause() external onlyOwner whenNotPaused {
        paused = true;
emit Pause();
}
```

Recommendation

The team should carefully manage the private keys of the owner's account.In the current context, the team should ensure that the ownership of the T oken contract is securely assigned to the `MultisigManager.sol` contract at the time of deployment. This setup ensures that all critical functions are protected by the multi-signature mechanism, thereby reducing the risk of a single point of failure or unauthorized access.

Additionally, it is advised to validate that the deployment and initialization process of the T oken contract does not inadvertently allow the contract to be owned by any account other than the intended `MultisigManager.sol` contract.

Suggested Solutions:

These measures, while improving security, do not eliminate the severity of the finding:

- Ensure that the signers of the multi-signature contract are distinct, independent entities with rigorously managed private keys.
- Ensure the multi-signature contract `MultisigManager.sol` is properly configured and operational from deployment.

- Ensure that the ownership of the Token contract is irrevocably assigned to the `MultisigManager.sol` contract and cannot be modified using the `transferOwnership()` method.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Token.sol#L118,132,144,170,171,172,189,207,208,227,228,313,321,329,409,536,537,560,561,562,599,600,618,619,632,633
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
uint public _totalSupply
uint _value
address _to
address _owner
address _from
address _spender
address[] calldata _tos
uint[] calldata _values
address _evilUser
address _clearedUser
address _blackListedUser
address _upgradedAddress
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	Token.sol#L64,411
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
owner = newOwner
upgradedAddress = _upgradedAdd
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	Token.sol#L2 MultisigManager.sol#L2 ManagedT oken.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^v0.8.30+;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested L TS (Long T erm Support) environment.

L23 - ERC20 Interface Misuse

Criticality	Minor / Informative
Location	Token.sol#L77,90,91,132,169,189,535,559,598
Status	Unresolved

Description

The ERC20 is a standard interface for tokens on the blockchain. It defines a set of functions and events that a contract must implement in order to be considered an ERC20 token. According to the ERC20 interface, the transfer function returns a bool value, which indicates the success or failure of the transfer. If the transfer is successful, the function returns true. If the transfer fails, the function returns false. The contract implements the transfer function without the return value.

```
function transfer(address to, uint value) external;
function transferFrom(address from, address to, uint value)
external;
function approve(address spender, uint value) external;

function transfer(address _to, uint _value) public virtual
override {
        balances[msg.sender] =
balances[msg.sender].sub(value);
        balances[_to] = balances[_to].add(_value);
        emit Transfer(msg.sender, _to, _value);
    }

...
```

Furthermore, the contract implements the `supportsInterface()` function which returns true when queried for the `interfaceID == 0x36372b07`. This is the ID for the standard interface as implemented in https://eips.ethereum.org/EIPS/eip-20, which is different from the one implemented in the contract. This discrepancy could lead to inconsistencies, as contracts interacting with `Token.sol` might expect a fully compliant implementation of the standard interface.

Recommendation

The incorrect implementation of the ERC20 interface could potentially lead to problems when interacting with the contract, as other contracts or applications that expect the ERC20 interface may not behave as expected.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
SafeMath	Library			
	mul	Internal		
	div	Internal		
	sub	Internal		
	add	Internal		
Ownable	Implementation	ERC173		
		Public	✓	-
	transferOwnership	External	✓	onlyOwner
ERC20Basic	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
ERC20	Interface	ERC20Basic		
	allowance	External		-
	transferFrom	External	✓	-
	approve	External	✓	-

ERC20Extended	Interface	ERC20		
	batchTransfer		✓	-
BasicToken	Implementation	Ownable, ERC20Basic		
		Public	✓	-
	transfer	Public	✓	-
	balanceOf	Public		-
StandardToken	Implementation	BasicToken, ERC20		
	transferFrom	Public	✓	-
	approve	Public	✓	-
	allowance	Public		-
ExtendedToken	Implementation	StandardToken, ERC20Extended		-
	batchTransfer	Public	✓	-
Pausable	Implementation	Ownable IPausable		
	pause	External	✓	onlyOwner whenNotPaused
	unpause	External	✓	onlyOwner whenPaused
BlackList	Implementation	Ownable, BasicToken, IBlackList		

	addBlacklist	External	✓	onlyOwner
	removeBlackList	External	✓	onlyOwner
	destroyBlackFunds	External	✓	onlyOwner
UpgradedStandardToken	interface	ERC20		
	transferByLegacy	External	✓	-
	transferFromByLegacy	External	✓	-
	approveByLegacy	External	✓	-
	batchTransferByLegacy	External	✓	-
Deprecateable	Implementation	Ownable, IDeprecatable		
		Public	✓	-
	deprecate	External	✓	onlyOwner
LegacyToken	Interface	ERC20, IDeprecatable		
	legacyBalance	External		-
	legacyAllowance	External		-
	emitTransfer	External	✓	-
	emitApproval	External	✓	-
Token	Implementation	Deprecateable, LegacyToken, Pausable, ExtendedToken, BlackList		

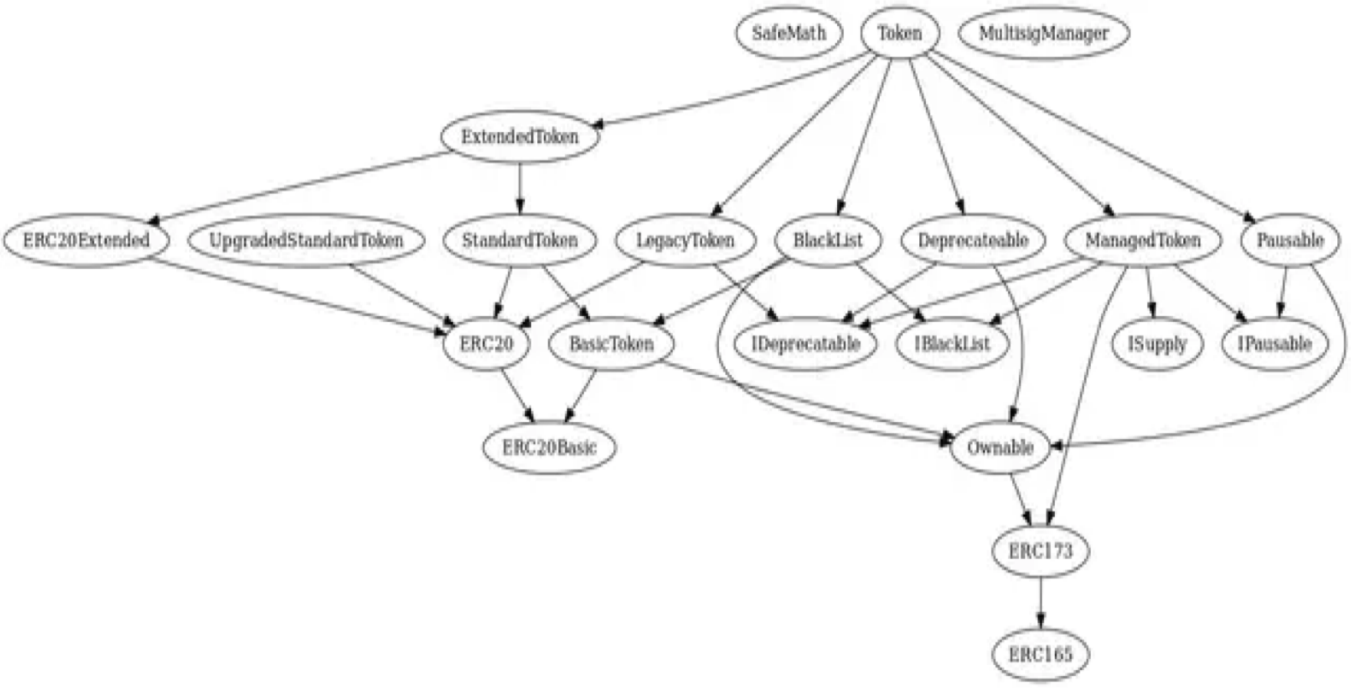
		ManagedToken		
		Public	✓	Ownable BasicToken
	supportsInterface	External		-
	transfer	Public	✓	whenNotPaused whenNotBlackListed
	transferFrom	Public	✓	whenNotPaused whenNotBlackListed
	balanceOf	Public		-
	approve	Public	✓	-
	allowance	Public		-
	batchTransfer	Public	✓	whenNotPaused whenNotBlackListed
	totalSupply	Public		-
	legacyBalance	External		onlyUpgraded
	legacyAllowance	External		onlyUpgraded
	emitTransfer	External	✓	onlyUpgraded
	emitApproval	External	✓	onlyUpgraded
	issue	External	✓	onlyOwner whenNotDeprecated
	redeem	External	✓	onlyOwner whenNotDeprecated
MultisigManager	Implementation			
	_addVotingAccount	Private	✓	

	_removeVotingAccount	Private	✓	
	_makeRequest	Private	✓	
	_approveRequest	Private	✓	
	getMinApprovals	Public		-
		Public	✓	-
	requestOwnerChange	External	✓	-
	approveOwnerChange	External	✓	-
	requestVotersListChange	External	✓	-
	approveVotersListChange	External	✓	-
	requestTokenPause	External	✓	-
	approveTokenPause	External	✓	-
	requestTokenUnpause	External	✓	-
	approveTokenUnpause	External	✓	-
	requestBlacklist	External	✓	-
	approveBlacklist	External	✓	-
	requestUnblacklist	External	✓	-
	approveUnblacklist	External	✓	-
	requestBlackFundsDestruction	External	✓	-
	approveBlackFundsDestruction	External	✓	-
	requestDeprecation	External	✓	-
	approveDeprecation	External	✓	-
	requestIssue	External	✓	-
	approveIssue	External	✓	-
	requestRedeem	External	✓	-
	approveRedeem	External	✓	-

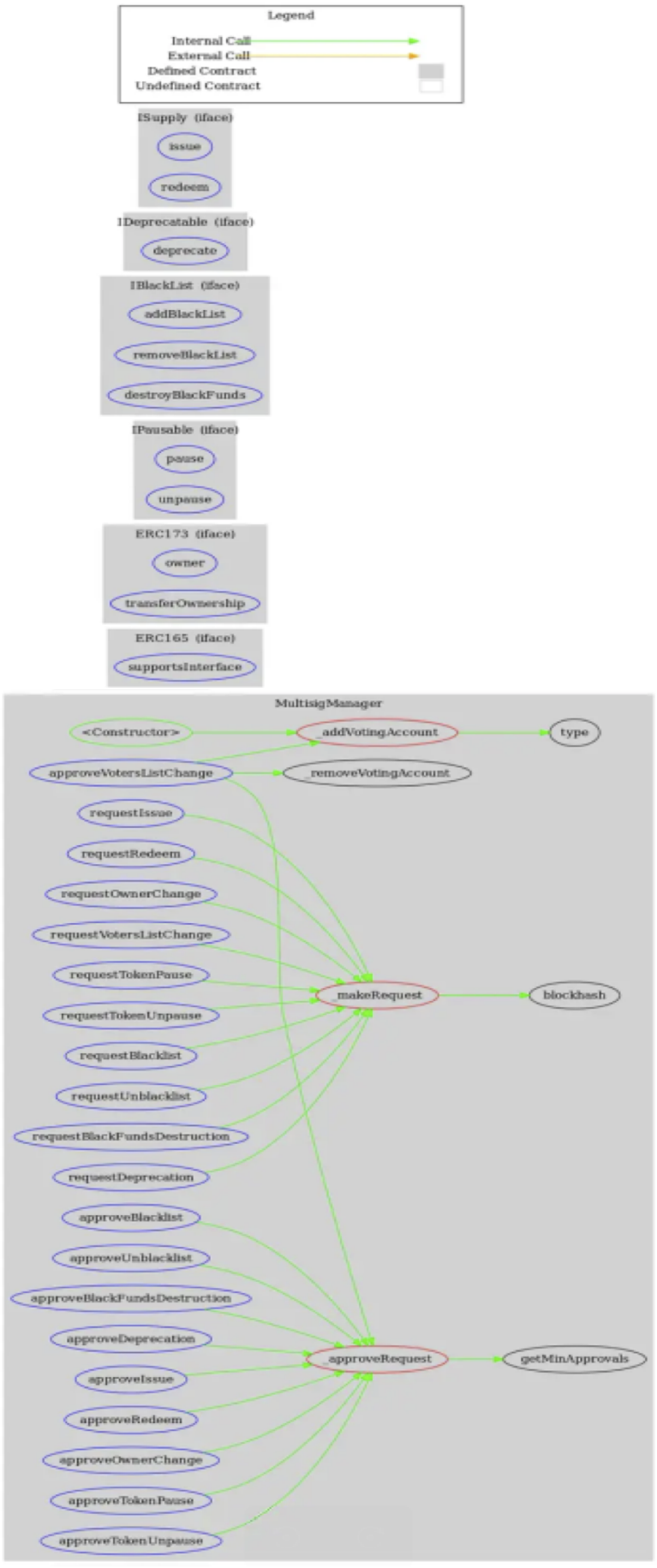
ERC165	Interface			
	supportsInterface	External		-
ERC173	Interface	ERC165		
	owner	External		-
	transferOwnership	External	✓	-
IPausable	Interface			
	pause	External	✓	-
	unpause	External	✓	-
IBlackList	Interface			
	addBlackList	External	✓	-
	removeBlackList	External	✓	-
	destroyBlackFunds	External	✓	-
IDeprecatable	Interface			
	deprecate	External	✓	-
ISupply	Interface			
	issue	External	✓	-
	redeem	External	✓	-

ManagedToken	Interface	ERC173, IPausable, IBlackList, IDeprecatable, ISupply		

Inheritance Graph



Inheritance Graph




LegacyToken (iface)
legacyBalance
legacyAllowance
emitTransfer
emitApproval
Deprecateable
<Constructor>
deprecate
onlyUpgraded
whenNotDeprecated
UpgradedStandardToken (iface)
transferByLegacy
transferFromByLegacy
approveByLegacy
batchTransferByLegacy
Pausable
whenNotPaused
whenPaused
pause
unpause
StandardToken
transferFrom
approve
allowance
_allowance
sub
ExtendedToken
batchTransfer
senderBalance
sub
BlackList
whenNotBlackListed
addBlackList
removeBlackList
destroyBlackFunds
balanceOf
ManagedToken (iface)
transferFrom
batchTransfer
balanceOf
transfer
approve
allowance
BasicToken
<Constructor>
tranafer
balanceOf
Token
<Constructor>
supportsInterface
transfer
transferFrom
balanceOf
approve
allowance
batchTransfer
totalSupply
legacyBalance
legacyAllowance
emitTransfer
emitApproval
issue
redeem
totalSupply
add
sub
ERC20Extended (iface)
batchTransfer
ERC20 (iface)
allowance
transferFrom
approve
ERC20Basic (iface)
totalSupply
balanceOf
transfer
Ownable
<Constructor>
onlyOwner
transferOwnership
SafeMath (lib)
mul
div
sub
add

Summary

MEPE is an innovative project with a rapidly growing community. The audit of its smart contracts revealed no compiler errors or critical issues. However, it is important to note that the contract owner has access to administrative functions that could potentially be misused if not properly managed. To mitigate this risk, the project has integrated a multi-signature contract intended to serve as the owner of the Token contract. This significantly enhances decentralisation by ensuring that all critical functions are secured through the multi-signature mechanism, which reduces the likelihood of a single point of failure or unauthorised access. Nevertheless, for effective decentralisation and security, it is crucial that the multi-signature wallet is correctly configured as the owner of the token contract from the deployment, and that the signers are independent entities with securely managed private keys.

This audit addresses security concerns, evaluates business logic, and suggests potential improvements to strengthen the overall robustness of the MEPE project.

Disclaimer

The information contained in this report does not constitute investment, financial, or trading advice. None of the content herein should be construed as such. This report may not be transmitted, disclosed, referred to, or relied upon by any person for any purpose, nor may copies be delivered to any third party other than the Company, without the prior written consent of AI Auditor.

This report shall not be construed as, nor does it constitute, an "endorsement" or "disapproval" of any particular project, company, or team. It shall not be regarded as an indication of the economic viability, value, or potential of any product, asset, or technology developed by a party that has engaged AI Auditor to perform a security or code assessment.

No warranty or guarantee, whether express or implied, is provided regarding the complete absence of vulnerabilities, defects, or bugs in the analyzed technology. Likewise, AI Auditor does not provide any assurances or representations regarding the business operations, models, legal standing, or regulatory compliance of the entities subject to this assessment.

This report is not intended to serve as a basis for investment decisions or as a recommendation to engage with or invest in any particular project or asset. It is the result of a professional security review process designed solely to assist clients in improving code quality and reducing risks associated with blockchain systems and cryptographic technologies.

Blockchain and cryptographic technologies inherently involve a high degree of operational and security risk. AI Auditor maintains that each organization and individual bears sole responsibility for conducting their own due diligence and maintaining ongoing security measures. While AI Auditor’s objective is to identify and help minimize potential attack vectors and vulnerabilities, no guarantee of security, functionality, or future performance of the analyzed technology is implied or provided.

The assessment services rendered by AI Auditor are subject to external dependencies, limitations, and ongoing methodological development. Accordingly, you acknowledge and agree that any access to or use of this report, including but not limited to any related services, materials, or findings, is undertaken entirely at your own risk, on an “as-is,” “where-is,” and “as-available” basis.

Cryptographic assets are experimental in nature and carry significant levels of technical, financial, and regulatory uncertainty. Assessment outcomes may include false positives, false negatives, or other unpredictable results. Furthermore, the services provided may rely on and interact with multiple layers of third-party infrastructure, each introducing its own potential risks and limitations.