



AUDIT REPORT

September 2025

For



Table of Content

Executive Summary	04
Number of Security Issues per Severity	06
Summary of Issues	07
Checked Vulnerabilities	08
Techniques and Methods	10
Types of Severity	12
Types of Issues	13
Severity Matrix	14
 High Severity Issues	15
1. Unintended Validator Addition via replaceValidator	15
 Medium Severity Issues	17
2. Signature Verification Mismatch in Burn Operations	17
 Low Severity Issues	19
3. Rescue Function Bypasses Blacklist/Freeze Restrictions	19
4.Blacklisted/Frozen Addresses Can Become Validators	21
5.Missing ERC20 Return Value Check in rescue and blacklisterFreezerOps Functions	23
6.Frozen Addresses Cannot Be Blacklisted	24
 Informational Issues	25
7.Missing Zero Address Checks in Role Initialization	25
8.Excessive Complexity and Lack of Documentation in Core Multi-Sig Functions	26



Functional Tests	27
Automated Tests	27
Closing Summary	28
Disclaimer	29



Executive Summary

Project Name	MNEE
Protocol Type	ERC20 Token
Project URL	https://www.mnee.io/
Overview	<p>The MNEE contract is an advanced ERC20-based stablecoin implementation designed for robust, multi-signature, and role-based governance. It is intended to provide a secure, compliant, and flexible digital asset with strong controls over minting, burning, pausing, blacklisting, freezing, and validator management.</p>
Audit Scope	<p>The scope of this Audit was to analyze the MNEE Smart Contracts for quality, security, and correctness.</p>
Source Code link	https://etherscan.io/address/0x89360ce7da79a5acd3fedc43778e4d6d28f09275#code
Contracts in Scope	MNEE
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	30th June 2025
Updated Code Received	2nd September 2025
Review 2	5th September 2025
Fixed In	https://github.com/mnee-xyz/blockchain/commit/eb19e7ab224cba7d818b4f8eb44d86a59089d276
Mainnet Address	https://etherscan.io/address/0x6D2E08839cEc8699565a86D046E64324826039c9#code

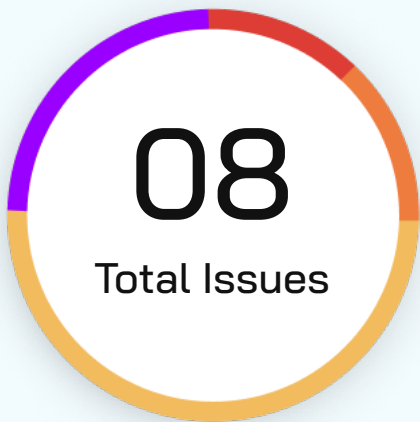


Verify the Authenticity of Report on QuillAudits Leaderboard:

<https://www.quillaudits.com/leaderboard>



Number of Issues per Severity



Critical	0(0.0%)
High	1 (12.5%)
Medium	1 (12.5%)
Low	4 (50.0%)
Informational	2 (25.0%)

		Severity				
		Critical	High	Medium	Low	Informational
Issues	Open	0	0	0	0	0
	Acknowledged	0	0	0	0	1
	Partially Resolved	0	0	0	0	0
	Resolved	0	1	1	4	1



Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Unintended Validator Addition via replaceValidator	High	Resolved
2	Signature Verification Mismatch in Burn Operations	Medium	Resolved
3	Rescue Function Bypasses Blacklist/Freeze Restrictions	Low	Resolved
4	Blacklisted/Frozen Addresses Can Become Validators	Low	Resolved
5	Missing ERC20 Return Value Check in rescue and blacklistFreezerOps Functions	Low	Resolved
6	Frozen Addresses Cannot Be Blacklisted	Low	Resolved
7	Missing Zero Address Checks in Role Initialization	Informational	Resolved
8	Excessive Complexity and Lack of Documentation in Core Multi-Sig Functions	Informational	Acknowledged



Checked Vulnerabilities

✓ Access Management

✓ Arbitrary write to storage

✓ Centralization of control

✓ Ether theft

✓ Improper or missing events

✓ Logical issues and flaws

✓ Arithmetic Computations
Correctness

✓ Race conditions/front running

✓ SWC Registry

✓ Re-entrancy

✓ Timestamp Dependence

✓ Gas Limit and Loops

✓ Exception Disorder

✓ Gasless Send

✓ Use of tx.origin

✓ Malicious libraries

✓ Compiler version not fixed

✓ Address hardcoded

✓ Divide before multiply

✓ Integer overflow/underflow

✓ ERC's conformance

✓ Dangerous strict equalities

✓ Tautology or contradiction

✓ Return values of low-level calls



✓ **Missing Zero Address Validation**

✓ **Private modifier**

✓ **Revert/require functions**

✓ **Multiple Sends**

✓ **Using suicide**

✓ **Using delegatecall**

✓ **Upgradeable safety**

✓ **Using throw**

✓ **Using inline assembly**

✓ **Style guide violation**

✓ **Unsafe type inference**

✓ **Implicit visibility level**

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.



Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.



Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

■ **Critical: Immediate and Catastrophic Impact**

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

■ **High (H): Significant Risk of Major Loss or Compromise**

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

■ **Medium (M): Potential for Moderate Harm Under Specific Circumstances**

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

■ **Low (L): Minor Imperfections with Limited Repercussions**

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

■ **Informational (I): Opportunities for Improvement, Not Immediate Risks**

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.



Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.







Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

		Impact		
		 High	 Medium	 Low
Likelihood	 High	Critical	High	Medium
	 Medium	High	Medium	Low
	 Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely.
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.



High Severity Issues

New Validators Can Be Added via replaceValidator Without Replacing Existing Ones

Resolved

Path

MNEE.sol

Function

replaceValidator

Description

The replaceValidator function in the MNEEcontract allows a group of authorized signers to “replace” an existing validator (minter, burner, pauser, or blacklist) with a new address. However, the function does not validate whether the old address being replaced is actually a current validator of the specified role. This opens a logic flaw: authorized signers can call replaceValidator using an arbitrary, non-validator address as the old address, thereby directly adding a new address as a validator without removing any legitimate one.

POC

```
/// This test demonstrates that a new minter can be added without removing any
legitimate minter.
/// The flaw: replaceValidator does not check if _old is actually a minter,
so any address can be used as _old.
/// Steps:
/// 1. Pick a non-minter address as _old (address(0xDEAD)).
/// 2. Use valid signatures from current minters to call replaceValidator
with this non-minter and a new address.
/// 3. Assert that the new address is now a minter, the non-minter remains
not a minter, and all original minters are still minters.
/// 4. Log the result for clarity.

function test_CanAddNewMinterWithoutRemovingAny() public {
    // Step 1: Pick a non-minter address as _old
    address nonMinter = address(0xDEAD); // not a minter
    address newMinter = address(0xBEEF);
    // Step 2: Use valid signatures from current minters
    address[3] memory signers_ = [minters[1], minters[2], minters[3]];
    bytes32 nonce = keccak256(abi.encodePacked("replace", nonMinter,
newMinter, block.timestamp, block.number));
    bytes[3] memory sigs = getValidSignatures(signers_, nonMinter,
newMinter, 0, uint8(MNEE.functionType.changeMinter), nonce);
    vm.prank(minters[1]);
    mnee.replaceValidator(nonMinter, newMinter,
MNEE.functionType.changeMinter, signers_, nonce, sigs);
    // Step 3: Assert the new address is a minter, non-minter remains not a
minter, and all original minters are still minters
    assertTrue(mnee.isMinter(newMinter), "New minter should be added");
    assertTrue(!mnee.isMinter(nonMinter), "Non-minter should remain not a
minter");
}
```



```
    for (uint I = 0; I < 4; I++) {
        assertTrue(mnee.isMinter(minters[I]), "Original minter should
remain");}
    // Step 4: Log the result for clarity
    console2.log("POC: New minter added without removing any legitimate
minter");
}
```

The test passes with

```
Ran 1 test for test/ReplaceValidator.t.sol:ReplaceValidatorTest
[PASS] test_CanAddNewMinterWithoutRemovingAny() (gas: 132049)
Logs:
  POC: New minter added without removing any legitimate minter

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.87ms (1.06ms CPU time)
Ran 1 test suite in 125.04ms (1.87ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total test)
```

Impact: HIGH

Validator set inflation: This allows the validator set to be expanded arbitrarily, contrary to protocol assumptions in all checks and verifications of there being 4 validators per role max at a time.

This introduces a direct threat to privileged functionality and governance integrity, which are central to protocol safety.

Likelihood: HIGH

No validation on old address: Exploitable in every call to replaceValidator.

Given this is a logic flaw in a core access control mechanism, and not reliant on specific conditions, the likelihood of misconfiguration or abuse is high.

Recommendation

Ensure the old address being replaced actually holds the intended role before assignment:

```
require(isMinter[_old], "Old address is not a minter");
```



Medium Severity Issues

Signature Verification Mismatch in Burn Operations

Resolved

Path

MNEE.sol

Path

mintBurnPauseUnpause

Description

The mintBurnPauseUnpausefunction contains a signature verification mismatch for burn operations. The comments explicitly state that burn operations should burn from the target address, but the implementation contradicts this by always burning from the redeemer address while still including the _target parameter in signature verification.

```
function mintBurnPauseUnpause(address _target, uint256 _amount,
functionType fType,address[requiredSignatures] calldata signers,
bytes[requiredSignatures] calldata signatures, bytes32 instanceIdentifier
) external { // ... signature verification includes _target parameter
    /.../
    if (!SigningLibrary.verify(
        signers[I],
        _target,           // Target is included in signature verification
        address(0),
        _amount,
        uint8(fType),
        instanceIdentifier,
        block.chainid,
        signatures[I]
    )) revert invalidSign();
    /.../
    // ... but burn operation ignores _target and always burns from redeemer
    if (fType == functionType.burn) {
        if (!isBurner[_msgSender()]) revert onlyCorrectValidator();
        super._burn(redeemer, _amount);
        emit TokensBurnt(_amount);
    }
}
```

The `SigningLibrary.verify()`function includes the `_target` parameter in the signature verification, but the burn operation completely ignores this value.



```
function verify(
    address _signer,
    address target, // ← Target parameter included in signature verification
    address target2,
    uint256 amount,
    uint8 functionType,
    bytes32 instanceIdentifier,
    uint256 chainId,
    bytes memory signature
) public pure returns (bool) {
    bytes32 messageHash = getMessageHash(
        target, target2, amount, functionType, instanceIdentifier, chainId
    );
    bytes32 ethSignedMessageHash = getEthSignedMessageHash(messageHash);
    return recoverSigner(ethSignedMessageHash, signature) == _signer;
}
```

This means:

- Any target address can be passed in the signature verification
- The burn operation is hardcoded to always burn from `redeemer` address
- The contract never validates that the signature target matches the actual burn target
- The signature authorizes burning from one address, but the contract burns from a different address

Impact: MEDIUM

This issue has medium impact because although it creates a mismatch between what the signature authorizes and what the contract actually executes - the burn functionality itself is still protected by access control

Likelihood: HIGH

This issue has HIGH likelihood of exploitation because:

- The mismatch occurs in every burn operation, not just edge cases
- There's no way to burn from a different address than redeemer
- This is a fundamental design issue in the burn mechanism
- The misleading documentation increases likelihood of incorrect usage
- Any target can be used with the same signature, making the target parameter meaningless

Recommendation

It is recommended to remove Target from Burn Signatures for burn operations, consider using a fixed value or address(0) in signature.

It will maintain the current centralized burn model while fixing the signature mismatch. Update documentation to clarify that burns always target the redeemer address for redemption purposes, and remove the misleading comment about target being "required for burning."



Low Severity Issues

Rescue Function Bypasses Blacklist/Freeze Restrictions

Resolved

Path

MNEE.sol

Function name

rescue

Description

The rescue function allows a trusted rescuer to recover ERC20 tokens accidentally stuck in the contract by transferring them to a specified recipient.

For example a user mistakenly sends USDT to the contract address. Those tokens are now stuck.

The project admin (with the rescuer role) can call:

```
rescue(USDT, 1000e6, userAddress);
```

This transfers 1000 USDT from the contract back to the user.

However, the function does not check whether the recipient – referred to as the requester – is blacklisted or frozen.

```
function rescue(  
    IERC20 token,  
    uint256 _amount,  
    address _requester  
) external {  
    if (!(rescuer == _msgSender())) revert onlyCorrectValidator();  
    if (address(token) == address(0)) revert zeroAddress();  
    if (_amount == 0) revert invalidAmt();  
    if (_requester == address(0)) revert zeroAddress();  
    token.transfer(_requester, _amount);  
    emit FundsRescued(address(token), _amount, _requester);  
}
```

The rescue function calls `token.transfer()` directly on external ERC20 tokens, which bypasses the MNEE contract's `_beforeTokenTransfer` hook that enforces blacklist and freeze restrictions.



```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    if (!isBlacklistFreezer[msg.sender]) {
        if (blacklisted[from]) revert blacklistedAddress();
        if (frozen[from]) revert frozenAddress();
    }
    if (blacklisted[to]) revert blacklistedAddress();
    if (paused()) revert tokenPaused();
    super._beforeTokenTransfer(from, to, amount);
}
```

This creates a logical inconsistency. While the contract may correctly prevent blacklisted or frozen users from receiving MNEE tokens (the native to the contract) — through enforcement in hook `_beforeTokenTransfer` — those same restrictions do not apply when rescuing other ERC20 tokens. The rescue function bypasses the transfer hooks and directly transfers tokens to the requester without verifying their blacklist or freeze status.

As a result, a user who is blacklisted or frozen would be correctly blocked from receiving MNEE tokens, but could still receive any other ERC20 token from the contract via the rescue function.

Impact: Low

Only affects non-core ERC20 tokens. MNEE transfers remain protected, and the value of rescued external tokens is typically low.

Likelihood: Low

Only callable by a trusted rescuer. Rescue actions are rare and usually reviewed manually, making accidental misuse unlikely.

Recommendation

Add explicit blacklist/freeze checks in the rescue function:

```
if (blacklisted[_requester]) revert blacklistedAddress();
if (frozen[_requester]) revert frozenAddress();
```



Blacklisted/Frozen Addresses Can Become Validators

Resolved

Path

MNEE.sol

Function name

`replaceValidator`

Description

The `replaceValidator` function allows a validator (minter, burner, pauser, or blacklister/freezer) to be replaced with an address that is currently blacklisted or frozen. This enables addresses that are supposed to be restricted from participating in the system to gain privileged roles.

```
function replaceValidator(
    address _old,
    address _new,
    functionType fType,
    address[requiredSignatures] calldata signers,
    bytes32 instanceIdentifier,
    bytes[requiredSignatures] calldata signatures
)
external{
    // ... signature verification ...
    if (fType == functionType.changeMinter) {
        if (!isMinter[_msgSender()]) revert onlyCorrectValidator();
        isMinter[_old] = false;
        isMinter[_new] = true; // ← No check if _new is blacklisted/frozen
        emit ChangeMinter(_old, _new);
    }
    // ... similar for other roles ...
}
```

The function verifies signatures and role eligibility, but does not check:

`blacklisted[_new]``frozen[_new]`

This means:

Blacklisted addresses can gain validator privileges

Frozen addresses can gain validator privileges

Restricted addresses can sign privileged operations Governance controls are undermined

Impact: Low

- While this undermines the spirit of blacklist/freeze, it doesn't directly let the address act maliciously.
- Real damage requires collusion of the multisig, which is already a higher-trust layer.
- The issue reflects a governance inconsistency, not an immediate threat to funds or protocol operations.



Likelihood: Low

- The action is gated behind a governance process (e.g., validator replacement proposal).
- Requires either negligence or malicious intent from multiple parties.
- The incentive

Recommendation

Add explicit blacklist/freeze checks in the replaceValidator function:

```
if (blacklisted[_requester]) revert blacklistedAddress();  
if (frozen[_requester]) revert frozenAddress();
```



Missing ERC20 Return Value Check in rescue and blacklistFreezerOps Functions

Resolved

Path

MNEE.sol

Function name

`replaceValidator`

Description

Both the rescue and blacklistFreezerOps functions perform ERC20 token transfers using the transfer() call. However, neither function checks the return value of this transfer operation.

According to the ERC20 standard, the transfer() function returns a boolean indicating whether the operation succeeded. Failing to check this return value may lead to scenarios where:

- The transfer silently fails (e.g., due to insufficient balance or a non-compliant token)
- The contract continues execution assuming success, potentially misleading users or breaking logical assumptions

This behavior can create hidden failures and inconsistent state, especially in recovery (rescue) or administrative (blacklist/unblacklist) operations.

Impact: Low

While not critical on its own, this issue can lead to silent failures, particularly with non-compliant ERC20 tokens. In rescue, it may leave users without expected funds, and in blacklistFreezerOps, it may falsely imply success in enforcement actions involving transfers.

If a token behaves unexpectedly (e.g., Tether's non-standard implementation), this could create subtle operational failures or trust issues.

Likelihood: Low

Most widely used ERC20 tokens are well-behaved and return true on successful transfer. However, the lack of a check still poses a best-practice deviation and will lead to silent failures in those functions.

Recommendation

Always verify the return value of the transfer function:

```
bool success = token.transfer(_requester, _amount);  
require(success, "Token transfer failed");
```

This ensures only successful transfers continue execution and protects the contract from interacting incorrectly with non-compliant or malfunctioning tokens.



Frozen Addresses Cannot Be Blacklisted

Resolved

Path

MNEE.sol

Function name

`replaceValidator`

Description

The `blacklistFreezerOps` function is responsible for applying enforcement actions such as blacklisting, freezing, delisting, unfreezing, confiscation, and burning of holdings. However, the logic prevents frozen addresses from being blacklisted. Specifically, in the case where `fType == functionType.blacklist`, the function checks:

```
if ((blacklisted[_address]) || (frozen[_address])) revert BLorF();
```

This means if an address is already frozen, any attempt to blacklist it will revert. As a result, there is no way to escalate the enforcement from a temporary freeze to a more permanent blacklist status.

This can create inconsistencies in enforcement policy. For example, if an address is frozen as an immediate mitigation, but later needs to be blacklisted due to further evidence or policy changes, the protocol will prevent this progression. This is counterintuitive in most compliance or security frameworks, where escalation from freeze to blacklist is a natural and expected flow.

Impact: Low

since frozen addresses are still restricted, but it may result in operational or governance inefficiencies.

Likelihood: Low

especially in systems where manual or automated escalations from freeze to blacklist are expected.

Recommendation

It is recommended to allow blacklisting of already frozen addresses by modifying the condition to only check `blacklisted[_address]`:

```
if (blacklisted[_address]) revert alreadyBL();
```

This would make blacklisting independent of freeze status, while still preventing duplicate blacklist attempts.



Informational Issues

Missing Zero Address Checks in Role Initialization

Resolved

Path

MNEE.sol

Path

`initialize`

Description

The initialize function in MNEE.sol accepts arrays of addresses to assign critical roles such as minter, burner, pauser, and blacklister. However, the code does not check whether any of the addresses inside these arrays are the zero address. Specifically, inside the loop that assigns roles, entries like `_minters[I]`, `_burners[I]`, `_pausers[I]`, and `_blacklisters_freezers[I]` are directly used without verifying they are non-zero.

```
for (I = 0; I < _minters.length; I++) {
    isMinter[_minters[I]] = true;
    isBurner[_burners[I]] = true;
    isPauser[_pausers[I]] = true;
    isBlacklistFreezer[_blacklisters_freezers[I]] = true;
```

This means an address like `address(0)` could be assigned a privileged role if mistakenly passed, which is not desirable. Although this does not introduce a direct exploit path, it reflects poor validation and opens the door for misconfigurations that are hard to track.

Impact: Low

The impact is low because `address(0)` cannot take action on its own, but it still represents bad practice.

Likelihood: Medium

since it's easy to misconfigure the initializer during deployment, especially when passing arrays manually or via scripts.

Recommendation

It is recommended to explicitly check that each address is non-zero before assigning roles, for example: `require(_minters[I] != address(0), "zero address");`

This should be repeated for each corresponding role array to ensure proper initialization safety.



Excessive Complexity and Lack of Documentation in Core Multi-Sig Functions

Acknowledged

Description

The functions `blacklistFreezerOps`, `mintBurnPauseUnpause`, and `replaceValidator` are central to the security and governance of the `MNEE` contract. However, these functions are highly complex, densely packed with logic, and lack sufficient inline documentation or clear structure. This results in code that is extremely difficult to read, understand, and audit. For example:

```
@dev function for a using blacklisting, delisting, freezing, unfreezing,
confiscation and holdings burning. requires 3 valid signs from 3 valid signers
* @param _address address target for blacklisting, delisting, freezing,
unfreezing, confiscation and holdings burning
* @param _to address receipient in case of confiscation
* @param _amount amount required for confiscation or burning holding.
inconsequential in case of blacklisting-delisting or freezing-unfreezing
* @param fType function code to be carried out
* @param signers signer addresses passed of the particular role
* @param signatures signatures approving this change
*/
function blacklistFreezerOps(
    address _address,
    address _to,
    functionType fType,
    uint256 _amount,
    address[requiredSignatures] calldata signers,
    bytes[requiredSignatures] calldata signatures,
    bytes32 instanceIdentifier
)
external {
    require(signers[0] != signers[2], "Signers must be unique");
    require(!instanceNonces[instanceIdentifier], "Invalid uuid");
    instanceNonces[instanceIdentifier] = true;
    if (!isBlacklistFreezer[_msgSender()]) revert onlyCorrectValidator();
    for (uint8 I = 0; I < requiredSignatures; I++) {
        if (!(isBlacklistFreezer[signers[I]])) revert invalidSigner();

        if(I >0)
            if(signers[I] == signers[i-1]) revert invalidSigner();
    if (
        !SigningLibrary.verify(
            signers[I],
            _address,
            _to,
            _amount,
            uint8(fType),
            instanceIdentifier,
            block.chainid,
            signatures[I]
        )
    ) revert invalidSign();
    }
}
```



```

if (fType == functionType.burnHoldings) {
    if (!(blacklisted[_address] || frozen[_address]))
        revert neitherBLnorF();
    if (_to != address(0)) revert notZeroAddress();
    super._burn(_address, _amount);
    emit HoldingsBurnt(_address, _amount);
} else if (fType == functionType.confiscate) {
    if (!(blacklisted[_address] || frozen[_address]))
        revert neitherBLnorF();
    if (_to == address(0)) revert zeroAddress();
    _transfer(_address, _to, _amount);
    emit FundsConfiscated(_address, _amount, _to);
} else if (fType == functionType.blacklist) {
    if ((blacklisted[_address]) || (frozen[_address])) revert BLorF();
    blacklisted[_address] = true;
    emit AccountBlacklisted(_address);
} else if (fType == functionType.freeze) {
    if ((blacklisted[_address]) || (frozen[_address])) revert BLorF();
    frozen[_address] = true;
    emit AccountFrozen(_address);
} else if (fType == functionType.unfreeze) {
    if (!frozen[_address]) revert notF();
    frozen[_address] = false;
    emit AccountUnfrozen(_address);
} else if (fType == functionType.delist) {
    if (!blacklisted[_address]) revert notBL();
    blacklisted[_address] = false;
    emit AccountDelisted(_address);
} else revert wrongFunction();
}

```

Each function handles multiple roles, actions, and edge cases within a single, lengthy code block. The logic for signature verification, role checks, and business actions is deeply nested and intertwined.

There are minimal or no inline comments explaining the purpose of key checks, the rationale for certain design decisions, or the intended flow of execution. The meaning of parameters and the expected behavior for each function type are not clearly described.

The combination of complexity and lack of documentation makes these functions nearly unreadable for new developers, auditors, or even future maintainers. This increases the risk of introducing or missing critical bugs and vulnerabilities.

Impact

The current state of these functions makes future maintenance, upgrades, or bug fixes error-prone and time-consuming.

New developers will struggle to understand or safely extend the contract.

Recommendation

Break down these functions into smaller, well-named internal functions, each handling a single responsibility (e.g., signature verification, role checks, business logic).

Add comprehensive inline comments and function-level documentation explaining the purpose, parameters, and expected behavior for each code path.

Use modifiers for common checks (e.g., role validation, signature validation) to improve clarity and reduce repetition.

Functional Tests

The tests written by the QuillAudits team for the 'MNEE' contract comprehensively cover its core security and governance mechanisms. They verify correct multi-signature enforcement for all privileged actions (minting, burning, pausing, blacklisting, freezing, validator replacement), ensure that replay attacks are prevented, and check that role-based restrictions are properly enforced. Additionally, the tests include edge cases and proof-of-concept scenarios for known logic flaws, such as the ability to add a new validator without removing an existing one, and confirm that the contract correctly blocks invalid or unauthorized operations.

Automated Tests

Some significant issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Replay Attack Test:

Ensures that signatures and nonces cannot be reused to repeat privileged actions (e.g., `test_SignatureReplayAttack`).

Role Confusion/Privilege Escalation Test:

Checks that only the correct roles can perform sensitive actions, and that signatures from one role cannot be used for another (e.g., `test_RoleConfusionVulnerability`, `test_PauseUnpauseRoleConfusion`).

Logic Flaw/Bypass Test:

Demonstrates that a new validator can be added without removing an old one due to a missing check (e.g., `test_CanAddNewMinterWithoutRemovingAny`).

Edge Case/State Violation Test:

Verifies that forbidden state transitions are blocked, such as blacklisting a frozen address (`test_CannotBlacklistFrozenAddress`). These tests are critical for identifying and demonstrating real-world threats and attack vectors, helping to ensure the contract's security before deployment.



Closing Summary

In this report, we have considered the security of the MNEE smart contract system. We performed our audit according to a standard procedure involving manual code review, logic validation, and assessment of critical operations such as access control, multisig coordination, and enforcement mechanisms.

A total of 8 issues were identified:

High Severity Issues (1)

Unintended Validator Addition via `replaceValidator` – A critical logic flaw allows adding new validators without properly removing an existing one, undermining validator governance.

Medium Severity Issues (1)

Signature Verification Mismatch in Burn Operations – Inconsistencies in expected signer roles may lead to verification failures or unauthorized burns.

Low Severity Issues (3)

- Rescue Function Bypasses Blacklist/Freeze Restrictions
- Blacklisted/Frozen Addresses Can Become Validators
- Missing ERC20 Return Value Check in `rescue` and `blacklistFreezerOps` Functions

Informational Severity Issues (2)

- Missing Zero Address Checks in Role Initialization
- Excessive Complexity and Lack of Documentation in Core Multi-Sig Functions

These findings highlight concerns primarily around validator management, signature enforcement, and code robustness. In the end, MNEE Team Resolved all Issues and Acknowledged one issue.



Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.



About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers.

With seven years of expertise, we've secured over 1400 projects globally, averting over \$3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

**7+**

Years of Expertise

1M+

Lines of Code Audited

50+

Chains Supported

1400+

Projects Secured

Follow Our Journey



AUDIT REPORT

September 2025

For



Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillaudits.com