QuillAudits

AUDIT REPORT

August 2025

For

maicrotrader

Table of Content

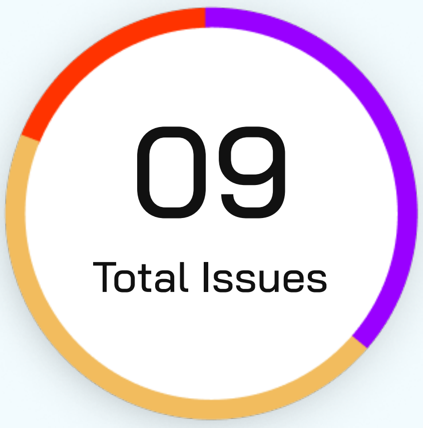
Executive Summary

Project Name	MaicroTrader
Protocol Type	Vault
Project URL	https://www.maicrotrader.com/
Overview	Maicrotrader Vault is an ERC-4626–style vault for USDC on Arbitrum, designed for off-chain NAV and share calculations. It supports: • Deposit (with or without EIP-2612 permit) • Off-chain share minting • Redemption with customizable cooldown & admin approval • Clean, minimal on-chain logic
Audit Scope	The scope of this Audit was to analyze the Maicrotrader Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/maicrotrader/maicrotrader-vault-contract
Branch	Main
Contracts in Scope	contracts/MaicrotraderVault.sol
Commit Hash	d9dc9b58b4c8d79901b3ea21f9b0ecc4443343bd
Language	Solidity
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	30th July 2025 - 4th August 2025
Updated Code Received	26th August 2025
Review 2	26th August 2025
Fixed In	fd02a58c62ff42833faab0dd63211c0e0b404834

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	0 (0%)
🟥	High	1 (11.1%)
🟧	Medium	0 (0%)
🟨	Low	4 (44.4%)
🟪	Informational	4 (44.4%)

Severity

Issues

	🟥 Critical	🟥 High	🟧 Medium	🟨 Low	🟪 Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	1	0	4	4

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Shares Burned Prematurely in requestRedeem Function	High	Resolved
2	Missing Event Emission on Cooldown Update	Low	Resolved
3	Missing ERC20 Return Value Check	Low	Resolved
4	CEI Pattern Violations	Low	Resolved
5	Potential Reentrancy in executeRedeem()	Low	Resolved
6	Missing Zero Address Checks in Role Initialization	Informational	Resolved
7	Gas Inefficient String Messages in Require Statements	Informational	Resolved
8	Poor Variable Naming	Informational	Resolved
9	Floating Pragma	Informational	Resolved

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

Likelihood \ Impact	🟥 High	🟧 Medium	🟨 Low
🟥 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

Shares Burned Prematurely in requestRedeem Function

Resolved

Path

MaicrotraderVault.sol

Function

`requestRedeem()`

Description

The **`MaicrotraderVault`** contract burns user shares immediately in the **`requestRedeem()`** function before any guarantee that the redemption will be processed or assets will be provided. This creates a vulnerability where users can permanently lose their shares without receiving any assets in return if the **`queueRedeem()`** is never called or if the redemption process fails.

```
/// @notice Request redemption by burning shares (off-chain assets compute)
function requestRedeem(uint256 shares) external {
    _burn(msg.sender, shares);  //<= Shares get burned
    uint256 reqId = nextRequestId;
    redeemRequests[reqId].user = msg.sender;
    userRedeemIds[msg.sender].push(reqId);
    emit RedeemRequested(reqId, msg.sender, shares);
    unchecked {
        nextRequestId++; //@done CEI pattern not followed
    }
}
```

The current implementation burns shares in **`requestRedeem`**, then requires the admin to queue the redemption, and finally allows the user to execute the redemption. If the admin never queues the redemption or if there are issues with the queueing process, the user's shares are permanently lost without any recourse or refund mechanism.

Impact: HIGH

The impact is high because it can lead to significant material loss of user assets, as shares represent ownership in the vault and their premature destruction results in permanent loss of value without any compensation or recovery mechanism

Likelihood: HIGH

The likelihood of this issue occurring is high since the current implementation will always burn shares immediately upon request, and any failure in the subsequent queueing or approval process will result in permanent share loss.

Recommendation

It is recommended to modify the redemption flow to lock shares instead of burning them immediately. The **`requestRedeem()`** function should transfer shares to the vault (locking them) rather than burning them, and the shares should only be burned when the admin calls **`queueRedeem`** and confirms the redemption will be processed or the burn can happen in **`executeRedeem()`**. This ensures users do not lose their shares until there is a guarantee that their redemption will be fulfilled.

For example:

```
struct RedeemInfo {
   address user;
   uint256 assets;
   uint256 sharesToBurn; //<= New field
   uint40 requestedAt;
   bool approved;
   bool executed


function requestRedeem(uint256 shares) external {
   if (shares == 0) revert ZeroShares();
   if (balanceOf(msg.sender) < shares) revert InsufficientShares();

   // Lock shares in vault instead of burning
   _transfer(msg.sender, address(this), shares);

   uint256 reqId = nextRequestId;
   redeemRequests[reqId].user = msg.sender;
   redeemRequests[reqId].sharesLocked = shares; // Track locked shares
   userRedeemIds[msg.sender].push(reqId);

   emit RedeemRequested(reqId, msg.sender, shares);
   nextRequestId++;
}


function queueRedeem(uint256 requestId, uint256 assets) external onlyOwner {
   RedeemInfo storage r = redeemRequests[requestId];
   if (r.requestedAt != 0) revert AlreadyQueued();
   if (r.sharesLocked == 0) revert NoSharesLocked();

   bool needsApproval = assets > ADMIN_THRESHOLD;
   r.assets = assets;
   r.requestedAt = uint40(block.timestamp);
   r.approved = !needsApproval;

   // Burn shares only when confirming redemption
   _burn(address(this), r.sharesLocked);
   r.sharesLocked = 0; // Clear locked shares

   emit RedeemQueued(requestId, assets, needsApproval);
}
```

```
function executeRedeem(uint256 requestId) external {
   RedeemInfo storage r = redeemRequests[requestId];
   if (r.requestedAt == 0 || r.executed) revert InvalidRequest();
   if (!r.approved) revert NotApproved();
   if (block.timestamp < r.requestedAt + cooldown) revert CooldownNotMet();
   if (totalAssets() < r.assets) revert InsufficientUSDC();

   // Transfer USDC to user
   IERC20(address(asset())).transfer(r.user, r.assets);

   r.executed = true;
   emit RedeemExecuted(requestId, r.user, r.assets);
}


function cancelRedeemRequest(uint256 requestId) external {
   RedeemInfo storage r = redeemRequests[requestId];
   if (r.user != msg.sender) revert NotYourRequest();
   if (r.requestedAt != 0) revert AlreadyQueued();
   if (r.sharesLocked == 0) revert NoSharesLocked();

   // Return locked shares to user
   _transfer(address(this), msg.sender, r.sharesLocked);

   // Clear the request
   delete redeemRequests[requestId];
   emit RedeemCancelled(requestId, msg.sender, r.sharesLocked);
}
```

Alternative approach would be to move the burn of shares next to the approval, which is a little more complex in terms of implementation as scenarios vary for amounts larger and smaller than ADMIN_THRESHOLD.

For example:

```
struct RedeemInfo {
   address user;
   uint256 assets;
   uint256 sharesToBurn; // New field
   uint40 requestedAt;
   bool approved;
   bool executed;
}
function requestRedeem(uint256 shares) external {
   if (shares == 0) revert ZeroShares();
   if (balanceOf(msg.sender) < shares) revert InsufficientShares();

   // Lock shares in vault instead of burning
   _transfer(msg.sender, address(this), shares);

   uint256 reqId = nextRequestId;
   redeemRequests[reqId].user = msg.sender;
   redeemRequests[reqId].sharesToBurn = shares; // Track shares to burn
   userRedeemIds[msg.sender].push(reqId);

   emit RedeemRequested(reqId, msg.sender, shares);
   nextRequestId++;
}
```

```
function queueRedeem(uint256 requestId, uint256 assets) external onlyOwner {
   RedeemInfo storage r = redeemRequests[requestId];
   if (r.requestedAt != 0) revert AlreadyQueued();
   if (r.sharesToBurn == 0) revert NoSharesToBurn();

   bool needsApproval = assets > ADMIN_THRESHOLD;
   r.assets = assets;
   r.requestedAt = uint40(block.timestamp);
   r.approved = !needsApproval;

   // Don't burn shares yet - wait for executeRedeem
   emit RedeemQueued(requestId, assets, needsApproval);
}


function executeRedeem(uint256 requestId) external {
   RedeemInfo storage r = redeemRequests[requestId];
   if (r.requestedAt == 0 || r.executed) revert InvalidRequest();
   if (!r.approved) revert NotApproved();
   if (block.timestamp < r.requestedAt + cooldown) revert CooldownNotMet();
   if (totalAssets() < r.assets) revert InsufficientUSDC();
   if (r.sharesToBurn == 0) revert NoSharesToBurn();

   // Burn shares at the moment of execution
   _burn(address(this), r.sharesToBurn);
   r.sharesToBurn = 0; // Clear shares to burn

   // Transfer USDC to user
   IERC20(address(asset())).transfer(r.user, r.assets);

   r.executed = true;
   emit RedeemExecuted(requestId, r.user, r.assets);
}


function cancelRedeemRequest(uint256 requestId) external {
   RedeemInfo storage r = redeemRequests[requestId];
   if (r.user != msg.sender) revert NotYourRequest();
   if (r.requestedAt != 0) revert AlreadyQueued();
   if (r.sharesToBurn == 0) revert NoSharesToBurn();

   // Return locked shares to user
   _transfer(address(this), msg.sender, r.sharesToBurn);

   // Clear the request
   delete redeemRequests[requestId];
   emit RedeemCancelled(requestId, msg.sender, r.sharesToBurn);
}
```

Low Severity Issues

Potential Reentrancy in executeRedeem()

Resolved

Path

MaicrotraderVault.sol

Function Name

`executeRedeem()`

Description

The reentrancy vulnerability in the **`MaicrotraderVault`** contract allows attackers to extract more funds than they should be entitled to. This vulnerability occurs in the **`executeRedeem()`** function, which is responsible for processing user redemption requests and transferring tokens back to users.

Take into consideration that the documentation only considers USDC token as vault currency but USDC is never hardcoded and thus this issue is low severity - as this vulnerability is only possible if the request asset is set to non-standard/malicious ERC-20 token (not USDC) that has a **`IERC20Receiver(to).onERC20Received(…);`** call in the **`transfer()`** function.

```
   /// @notice After `cooldown`, send exact USDC `assets` to user
   function executeRedeem(uint256 requestId) external {
       RedeemInfo storage r = redeemRequests[requestId];
       require(r.requestedAt != 0 && !r.executed, "invalid request");
       require(r.approved, "not approved");    //   <= REQUIRE (check)
       require(block.timestamp >= r.requestedAt + cooldown, "cooldown");
       require(totalAssets() >= r.assets, "insufficient USDC");


       // direct transfer of exact USDC amount
       IERC20(address(asset())).transfer(r.user, r.assets);//<= TRANSFER
(interaction)
        r.executed = true; //<= STATE UPDATE (effect)


       emit RedeemExecuted(requestId, r.user, r.assets);
   }
```

The problem lies in the order of operations within the **`executeRedeem()`** function. The function first performs all the necessary checks to ensure the request is valid and approved, then immediately jumps to interactions and calls the **`transfer()`** function to send assets to the user. However, the function only updates the state (the executed flag) after the transfer is complete.

This creates a dangerous window of opportunity. When the vault calls the transfer function to send tokens to the user, if the user is a smart contract, the transfer can trigger a callback function called **`onERC20Received()`** on the recipient contract.

Impact: HIGH

The reentrancy vulnerability in the **`MaicrotraderVault`** contract represents a high-impact security flaw that can lead to significant material loss of assets in the protocol. This vulnerability can be exploited repeatedly, potentially draining the vault of all its funds.The root cause is a violation of the Checks-Effects-Interactions pattern, which is a fundamental security principle in smart contract development. The vulnerability allows attackers to extract an arbitrary amount of tokens not limited to their redemption requests, effectively stealing funds from the vault's reserves.

Likelihood: LOW

The likelihood of this vulnerability being exploited is low, as it depends on the integration of a non-standard or malicious ERC-20 token that deliberately implements callback logic such as **`onERC20Received()`**. In the current setup—where the standard tokens like USDC is intended to be used — exploitation is not feasible. However, the issue is still worth reporting, as the contract's design assumes safe token behavior and could be exposed if the asset is ever replaced with a token that enables reentrancy.

Recommendation

It is recommended to modify the **`executeRedeem()`** function to follow the Checks-Effects-Interactions (CEI) pattern and include a **`nonReentrant`** modifier from OpenZeppelin's **`ReentrancyGuard`** contract.

For example:

```
 import "@openzeppelin/contracts/security/ReentrancyGuard.sol"; //<= New OZ
import
     /… /
/// @notice After `cooldown`, send exact USDC `assets` to user
   function executeRedeem(uint256 requestId) external nonReentrant{//<= Added
modifier
       RedeemInfo storage r = redeemRequests[requestId];


       // CHECKS
       require(r.requestedAt != 0 && !r.executed, "invalid request");
       require(r.approved, "not approved");
       require(block.timestamp >= r.requestedAt + cooldown, "cooldown");
       require(totalAssets() >= r.assets, "insufficient USDC");


       // EFFECTS
       r.executed = true;


       // INTERACTIONS
       asset.transfer(r.user, r.assets);


       emit RedeemExecuted(requestId, r.user, r.assets);
   }
```

The corrected implementation should first perform all validation checks, then update the state/set effects by setting the executed flag to true, and finally perform the external interaction by calling the **`transfer()`** function.

Missing Event Emission on Cooldown Update

Resolved

Path

MaicrotraderVault.sol

Function Name

`setCooldown()`

Description

The `setCooldown()` function in the `MaicrotraderVault` contract allows the owner to modify the cooldown period without emitting any events to notify users of this important state change.

```
/// @notice Owner sets new cooldown period
function setCooldown(uint256 newCooldown) external onlyOwner {
    cooldown = newCooldown;
}
```

The function directly updates the cooldown variable without any event emission, making it difficult for users and external systems to track when and how the cooldown period has been modified.

This lack of event emission creates transparency issues because users cannot easily monitor changes to the cooldown parameter, which directly affects their ability to execute redemption requests. The cooldown period is a critical parameter that determines how long users must wait after a redemption is queued before they can execute it, and changes to this value can significantly impact user experience and expectations.

Impact: LOW

The impact is low because it does not introduce security vulnerabilities or functional bugs, but it represents poor transparency and makes it difficult for users to track important governance decisions that affect their interactions with the contract.

Likelihood: MEDIUM

The likelihood of this issue causing problems is medium since the cooldown parameter affects all redemption operations and users rely on consistent behavior for their redemption timing.

Missing ERC20 Return Value Check

Resolved

Path

MaicrotraderVault.sol

Function Name

`executeRedeem()`

Description

The **`executeRedeem()`** function performs ERC20 token transfer using the transfer() call. However, the function fails to check 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)
- The contract continues execution assuming success, potentially misleading users or breaking logical assumptions

This behavior can create hidden failures and inconsistent state.

Impact: LOW

This issue can lead to silent failures, particularly with non-compliant ERC20 tokens. It may leave users without expected funds, and it may falsely imply success.

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

It is recommended to verify the return value of the transfer() function:

```
bool success = IERC20(address(asset())).transfer(r.user, r.assets);
require(success, “Token transfer failed”);
```

This ensures only successful transfers continue execution.

CEI Pattern Violations

Resolved

Path

MaicrotraderVault.sol

Function Name

`executeRedeem()`, `mintShares()`, `requestRedeem()`

Description

The `MaicrotraderVault` contract contains multiple functions that violate the Checks-Effects-Interactions (CEI) pattern, which is an important security principle in smart contract development. The `mintShares()` function increments the `nextRequestId` counter after minting shares, and the `requestRedeem()` function increments the `nextRequestId` counter after burning shares, both of which violate the CEI pattern by performing state updates after external calls.

Although the failure to follow the CEI pattern in `mintShares()` and `requestRedeem()` does not pose a direct security threat—since they do not perform external calls that could enable a classic reentrancy attack—it still introduces unnecessary risk and deviates from best practices, potentially increasing the surface for future vulnerabilities as the code evolves

Impact: LOW

The CEI pattern violations in `mintShares` and `requestRedeem` functions represent a low-impact issue because these functions do not contain external calls that could be exploited for a classic reentrancy attacks. The violations involve internal OpenZeppelin functions rather than external contract interactions. While this violates the CEI pattern principle, it does not create any direct security vulnerabilities or exploit paths that could lead to loss of funds or manipulation of the contract's state.

Likelihood: LOW

The likelihood of this issue causing problems is low. Since there are no external interactions that could be exploited, the CEI pattern violations in these functions represent more of a code quality issue than a security concern.

Recommendation

It is recommended to follow the Checks-Effects-Interactions (CEI) pattern throughout the contract, even when the interactions are internal (e.g., calling other functions within the same contract or modifying internal structures).

The CEI pattern is a best practice that helps ensure predictable execution flow and reduces the risk of bugs or vulnerabilities—particularly reentrancy issues—by enforcing the following order:

1. **Checks** – Validate inputs and conditions.
2. **Effects** – Update internal state.
3. **Interactions** – Perform any logic that depends on or may alter other components, even internally.

Informational Issues

Missing Zero Address Checks in Role Initialization

Resolved

Path

MaicrotraderVault.sol

Function Name

`constructor()`

Description

The **`MaicrotraderVault`** constructor accepts an owner address parameter to assign critical administrative privileges through the **`Ownable`** modifier.

```
/// @param usdc_           Underlying USDC token (6 decimals)
/// @param name_           Name for the vault's share token
/// @param symbol_         Symbol for the vault's share token
/// @param owner_          Initial owner (admin)
/// @param initialCooldown Initial cooldown in seconds
constructor(
    IERC20Metadata usdc_,
    string memory name_,
    string memory symbol_,
    address owner_,
    uint256 initialCooldown
) ERC20(name_, symbol_) ERC4626(usdc_) Ownable(owner_) {
    nextRequestId = 1;
    cooldown = initialCooldown;
}
```

However, the code does not check whether the owner address is the z ero address. Specifically, the owner parameter is directly passed to the **`Ownable`** constructor without verifying it is non-z ero, as shown where **`Ownable(owner_)`** is called without any validation.

This means an address like address(0) could be assigned the owner role if mistakenly passed during contract deployment, 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 and impossible to fix as the contract is not upgradable.

Recommendation

It is recommended to explicitly check that the address is non-z ero before assigning roles, for example:

```
require(owner_ != address(0), Zero address);
```

Gas Inefficient String Messages in Require Statements

Resolved

Path

MaicrotraderVault.sol

Function Name

`queueRedeem(), approveRedeem(), executeRedeem(), redeem()`

Description

The MaicrotraderVault contract uses string messages in `require/revert` statements throughout the codebase, which is not optimal from a gas efficiency perspective. Multiple `require/revert` statements across various functions include string literals such as:

- `“Zero shares”,`
- `“invalid request”,`
- `“not approved”,`
- `“cooldown”,`
- `“insufficient USDC”,`
- `“already queued”,`
- `“invalid state”,`
- `“use requestRedeem”,`

which increase the bytecode size and gas costs compared to using custom errors. While string messages in require statements provide helpful debugging information, they consume additional gas because the string data must be stored in the bytecode.

Custom errors, introduced in **Solidity 0.8.4**, provide the same debugging benefits but with significantly lower gas costs since they only store a 4-byte selector instead of the full string.

Recommendation

It is recommended to replace all string-based require statements with custom errors for better gas efficiency. Define custom errors at the contract level and use them throughout the contract:

```
error ZeroShares();
error InvalidRequest();
error NotApproved();
error CooldownNotMet();
error InsufficientUSDC();
error AlreadyQueued();
error InvalidState();
error UseRequestRedeem();


// Replace require statements like:
// require(shares > 0, “Zero shares”);
// with:
if (shares == 0) revert ZeroShares()
```

Poor Variable Naming

Resolved

Path

MaicrotraderVault.sol

Function Name

`queueRedeem()`, `approveRedeem()`, `executeRedeem()`

Description

The `MaicrotraderVault` contract contains multiple functions that use single-letter variable names, which is a poor coding practice that reduces code readability and maintainability. Functions such as:

- `queueRedeem()`,
- `approveRedeem()`,
- `executeRedeem()`,

An in-code example:

```
     /…/
RedeemInfo storage r = redeemRequests[requestId];
       require(r.requestedAt == 0, "already queued");
       bool needsApproval = assets > ADMIN_THRESHOLD;
       r.assets = assets;
       r.requestedAt = uint40(block.timestamp);
       r.approved = !needsApproval;
     /…/
```

The variable r is used to reference the `RedeemInfo` struct. The single-letter variable names make the code difficult to understand, especially for new developers or auditors who need to quickly comprehend the logic flow.

Single-letter variable names violate coding best practices because they provide no semantic meaning about what the variable represents, making it harder to debug, maintain, and review the code.

Recommendation

It is recommended to replace all single-letter variable names with descriptive names that clearly indicate their purpose. For example, replace r with `redeemRequest`.

Floating Pragma

Resolved

Path

MaicrotraderVault.sol

Description

In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma **(e.g., ^0.8.28)** introduces uncertainty, as it allows contracts to be compiled with any version within a specified range. This can result in discrepancies between the compiler used in testing and the one used in deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The project currently uses floating pragma declarations (^0.8.28) in its Solidity contracts. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

Recommendation

It is recommended to lock the pragma version to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler version, preventing unexpected changes in behavior due to compiler updates. For example, instead of using ^0.8.28, explicitly define the version with pragma solidity 0.8.28;.

Before selecting a version, review known bugs and vulnerabilities associated with each Solidity compiler release. This can be done by referencing the official Solidity compiler release notes: **Solidity GitHub releases** → or **Solidity Bugs by Version** →. Choose a compiler version with a good track record for stability and security.

Functional Tests

Some of the tests performed are mentioned below:

- ✓ Basic functionality (deposits, mints, redemptions)
- ✓ Access control (owner-only functions)
- ✓ Error conditions (insufficient balances, invalid states)
- ✓ Integration scenarios (full workflows, multiple users)
- ✓ Configuration (cooldown settings)

Automated Tests

No major 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.

Closing Summary

In this report, we have considered the security of MaicroTrader. We performed our audit according to the procedure described above. We conducted both automated and manual testing, including scenario-based attacks, to identify potential vulnerabilities and deviations from best practices.

While several issues of varying severity were identified, the majority can be addressed with straightforward code changes and adherence to established development patterns such as Checks-Effects-Interactions (CEI) and defensive coding practices. We commend the development team for their overall clean and modular codebase, which facilitated the audit process and indicates a thoughtful approach to security.

It is recommended to address all findings, regardless of severity, and conduct a final review after remediation. Security is an ongoing process, and we encourage the team to implement continuous testing, monitoring, and secure development practices throughout the lifecycle of the project.

Disclaimer

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.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

August 2025

For

maicrotrader

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com