Meme Strategy
Audit Report

Fri Oct 17 2025

contact@bitslab.xyz https://twitter.com/scalebit_

ScaleBit

Meme Strategy Audit Report

1 Executive Summary

1.1 Project Information

Description	MEMS is an innovative DeFi protocol that combines automated meme token trading through a queue-based system, dynamic fee collection via a Uniswap V4 hook with 3-phase fee system , a buyback and burn mechanism that automatically uses profits to repurchase and burn MEMS tokens, and treasury operations that allocate 80% of fees to the strategy while 20% goes to the treasury as rake
Type	DeFi
Auditors	Alex,hyer,Light
Timeline	Thu Oct 09 2025 - Fri Oct 17 2025
Languages	Solidity
Platform	EVM Chains
Methods	Architecture Review, Unit Testing, Manual Review
Source Code	https://github.com/yokai-laboratory/mems-strategy
Commits	2ba2cff9a5e7bf2ff6332d1ad0b7d445aece3665 cc0f970fa3ad397b35602cea2e8087face56e5e7 d493c92f9b2b0b8a1113f073af9d97106a7daabd

1.2 Files in Scope

The following are the SHA1 hashes of the original reviewed files.

ID	File	SHA-1 Hash
MST	src/MemeStrategy.sol	5d42b5345345c44c00a7cf2f5b6c95c39ee37591
MSH	src/MemeStrategyHook.sol	2d2d44c73954b0445675d84791ebd93afa513443

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	9	8	1
Informational	4	4	0
Minor	4	4	0
Medium	1	0	1
Major	0	0	0
Critical	0	0	0

1.4 ScaleBit Audit Breakdown

ScaleBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Integer overflow/underflow
- Number of rounding errors
- Unchecked External Call
- Unchecked CALL Return Values
- Functionality Checks
- Reentrancy
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic issues
- Gas usage
- Fallback function usage
- tx.origin authentication
- Replay attacks
- Coding style issues

1.5 Methodology

The security team adopted the **"Testing and Automated Analysis"**, **"Code Review"** and **"Formal Verification"** strategy to perform a complete security test on the code in a way that is closest to the real attack. The main entrance and scope of security testing are stated in the conventions in the "Audit Objective", which can expand to contexts beyond the scope according to the actual testing needs. The main types of this security audit include:

(1) Testing and Automated Analysis

Items to check: state consistency / failure rollback / unit testing / value overflows / parameter verification / unhandled errors / boundary checking / coding specifications.

(2) Code Review

The code scope is illustrated in section 1.2.

(3) Audit Process

- Carry out relevant security tests on the testnet or the mainnet;
- If there are any questions during the audit process, communicate with the code owner in time. The code owners should actively cooperate (this might include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- The necessary information during the audit process will be well documented for both the audit team and the code owner in a timely manner.

2 Summary

This report has been commissioned by Meme Strategy to identify any potential issues and vulnerabilities in the source code of the MemeStrategy smart contract, as well as any contract dependencies that were not part of an officially recognized library. In this audit, we have utilized various techniques, including manual code review and static analysis, to identify potential vulnerabilities and security issues.

During the audit, we identified 9 issues of varying severity, listed below.

ID	Title	Severity	Status
MSH-1	Centralization Risk	Medium	Acknowledged
MSH-2	Redundant Code	Informational	Fixed
MST-1	`addOperator` and `removeOperator` Lack Duplicate State Checks	Minor	Fixed
MST-2	Missing Check to Prevent Redundant Router Updates	Minor	Fixed
MST-3	`addModule` and `removeModule` Functions are Missing Event Emissions	Minor	Fixed
MST-4	Missing Existence Checks in `addMemToken` and `removeMemeToken`	Minor	Fixed
MST-5	Inconsistent Revert Message in onlyOwnerOrOperator Modifier	Informational	Fixed

MST-6	Comment Description Inconsistent with Code Implementation	Informational	Fixed
MSV-1	Unused `purchaseId` Parameter Passed to `_addActivePurchase()` Function	Informational	Fixed

3 Participant Process

Here are the relevant actors with their respective abilities within the MemeStrategy Smart Contract :

Owner

- `addOperator` : Add operator address.
- `removeOperator` : Remove operator address.
- `renounceOwnership` : Permanently renounce ownership.
- `setFeeExemption` : Set fee exemption status for addresses, allowing specific modules to bypass fees.
- `disableFees` : Disable fee collection.
- `updateFeeTier` : Update single fee tier parameters.
- `updateAllFeeTiers` : Update all fee tier configurations at once.
- `emergencyWithdrawETH` : Emergency withdraw ETH funds from contract.
- `initializePoolAndAddLiquidity` : Initialize Uniswap V4 pool and add initial liquidity.
- `enableTrading` : Enable token trading.
- `addMemeToken` : Add other tokens to whitelist for trading.
- `removeMemeToken` : Remove tokens from whitelist.
- `updateTokenRouter` : Update router address for whitelisted tokens.
- `setExecutorReward` : Set executor reward amount.
- `setMaxSaleSlippage` : Set maximum slippage tolerance when selling tokens.
- `setMinForceSellBPS` : Set minimum return percentage for forced sales.
- `setMaxBuybackSlippage` : Set maximum slippage for MEMS token buybacks.
- `setHookAddress` : Update hook contract address.

- `addApprovedRouter` : Add approved router addresses for trading.
- `removeApprovedRouter` : Remove approved router addresses.
- `setMaxSinglePurchase` : Set maximum ETH amount for single purchase.
- `setMinSpikeThreshold` : Set minimum ETH threshold to trigger fee spikes.
- `setMaxQueueSize` : Set maximum size for purchase queue.
- `addModule` : Add module contracts to whitelist.
- `removeModule` : Remove module contracts from whitelist.
- `pause` : Pause all critical contract operations.
- `unpause` : Unpause and resume contract operations.
- `forceSell` : Force sell tokens for specific purchase ID.
- `forceSellToTreasury` : Force sell tokens to treasury.
- `forceBurn` : Manually execute buyback and burn of MEMS tokens.
- `emergencyWithdrawTokens` : Emergency withdraw ERC20 tokens from contract (excluding MEMS).
- `emergencyWithdrawETH` : Emergency withdraw ETH funds from contract.
- `fundModule` : Push ETH funds to whitelisted modules.

Operator

- `enableFees` : Enable fee collection system.
- `flushToStrategy` : Flush accumulated strategy fees to MemeStrategy contract.
- `flushRake` : Flush accumulated rake fees to treasury address.
- `flushAllFees` : Flush all pending fees at once.
- `updateFeeTier` : Update fee tier parameters.
- `updateAllFeeTiers` : Update all fee tiers.

- `enableTrading` : Enable token trading.
- `addMemeToken` : Add other tokens to whitelist.
- `removeMemeToken` : Remove tokens from whitelist.
- `queueMemePurchase` : Add purchase requests to queue.
- `forceBuyMeme` : Force immediate purchase.
- `clearQueue` : Clear specific queue ID purchase requests.
- `removeQueuedPurchase` : Remove specific purchase requests from queue.
- `executeQueuedPurchase` : Execute specific purchase requests from queue.

MemeStrategy Contract

- `notifyFirstSale` : Notify hook contract that first sale is completed, triggering Phase 2 activation.
- `triggerFeeSpike` : Trigger fee spike based on sale amount.
- `addFeesETH` : Receive ETH fees transferred from hook contract.

PoolManager

- `beforeSwap` : Called before swap to handle buy-side fees.
- `afterSwap` : Called after swap to handle sell-side fees.
- `unlockCallback` : Handle Uniswap V4 unlock callback to execute atomic swaps.

User

- `getCurrentFee` : Query current dynamic fees.
- `getPhaseInfo` : Get current phase information.
- `getSpikeInfo` : Get active fee spike information.
- `getTierForAmount` : Query corresponding fee tier based on ETH amount.
- `getPendingFees` : Get pending fee balances.

- `executeSale` : Execute token sale and immediately buyback MEMS when target price is reached.
- `executeSaleToTreasury` : Execute token sale to treasury.
- `receive` : Receive ETH deposits to contract.

Module

- `pullFundsFromTreasury` : Pull ETH funds from treasury to module contract.

4 Findings

MSH-1 Centralization Risk

Severity: Medium

Status: Acknowledged

Code Location:

src/MemeStrategyHook.sol#305,315,327,338,350,363,377,388,407,545,565,769,791,836;

src/MemeStrategyHook.sol#462,510,517,526,611,627,640,668,725,761,776,795,803,807,814,818,823,8

Descriptions:

Both `MemeStrategy.sol` and `MemeStrategyHook.sol` give the `Owner` (and in some cases `operator`) broad unilateral powers over funds, trading behavior, and economic parameters.

Key privileged capabilities include:

- Fund Withdrawal: `emergencyWithdrawETH`, `emergencyWithdrawTokens` and `emergencyWithdrawETH` allow the `Owner` to withdraw essentially all ETH and ERC20 held by the contracts.
- Trade Intervention: `forceBuyMeme` and `forceSell` allow `Owner/operator` to bypass normal queues and price checks to execute arbitrary trades.
- System Pause: `pause() / unpause()` permit immediate suspension or resumption of core protocol functions.
- Economic Parameter Control: `Owner` can change critical parameters (e.g., `executorReward`, `maxSaleSlippage`, `maxSinglePurchase`) that directly affect protocol behaviour and user outcomes.
- Module & Router Management: `addModule`, `addApprovedRouter`, `fundModule` allow adding/removing and funding external modules/routers — potentially directing funds to arbitrary contracts.

Impact

If the `Owner` or any `authorized operator` is compromised, or if these roles act maliciously, an attacker can:

- Drain protocol treasury and user funds.
- Manipulate markets by forcing trades or setting abusive fee schedules.
- Pause the protocol maliciously.
- Redirect funds to malicious modules or routers.

Suggestion:

Use multi-signature.

MSH-2 Redundant Code

Severity: Informational

Status: Fixed

Code Location:

src/MemeStrategyHook.sol#550 626

Descriptions:

In the `beforeSwap` and `afterSwap` hook implementations, the exact same preliminary checks exist:

```
if (!feesEnabled) { ... }

if (sender == address(STRATEGY) || feeExempt[sender]) { ... }
```

Both functions repeat these checks at their entry points, and they always yield the same result.

Suggestion:

Combine the conditions into a single-line check at the entry of both functions.

```
if (
    !feesEnabled ||
    sender == address(STRATEGY) ||
    feeExempt[sender]
) {
    return (...);
}
```

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MST-1 `addOperator` and `removeOperator` Lack Duplicate State Checks

Severity: Minor

Status: Fixed

Code Location:

src/MemeStrategy.sol#513-523;

src/MemeStrategyHook.sol#301-313

Descriptions:

In the current implementation, the logic of `addOperator` and `removeOperator` is as follows:

```
function addOperator(address _operator) external onlyOwner {
    if (_operator == address(0)) revert InvalidAddress();
    operators[_operator] = true;
    emit OperatorAdded(_operator);
}

function removeOperator(address _operator) external onlyOwner {
    operators[_operator] = false;
    emit OperatorRemoved(_operator);
}
```

However, both functions lack state duplication checks:

- `addOperator` does not verify whether `_operator` is already set to `true`. Adding the same address multiple times will not revert, but it will repeatedly emit the `OperatorAdded` event, causing inconsistency between the contract state and the event logs.
- `removeOperator` does not verify whether `_operator` is an existing operator. Removing an address that was never added will not revert either, leading to misleading `OperatorRemoved` events.

Suggestion:

Add duplicate state checks to both functions:

```
function addOperator(address _operator) external onlyOwner {
    if (_operator == address(0)) revert InvalidAddress();
    if (operators[_operator]) revert AlreadyOperator(); //  Prevent duplicate addition
    operators[_operator] = true;
    emit OperatorAdded(_operator);
}

function removeOperator(address _operator) external onlyOwner {
    if (!operators[_operator]) revert NotOperator(); // Prevent duplicate removal
    operators[_operator] = false;
    emit OperatorRemoved(_operator);
}
```

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MST-2 Missing Check to Prevent Redundant Router Updates

Severity: Minor

Status: Fixed

Code Location:

src/MemeStrategy.sol#634

Descriptions:

The `updateTokenRouter()` function allows the contract owner to update the V2 router address for a whitelisted token. However, it does not check whether the new router address (`_newRouter`) is the same as the current one.

```
function updateTokenRouter(
        address _token,
        address _newRouter
    ) external onlyOwner {
        if (_newRouter == address(0)) revert InvalidAddress();
        if (!approvedRouters[_newRouter]) revert RouterNotApproved();
        if (!whitelistedTokens[_token].isWhitelisted)
            revert TokenNotWhitelisted();

        whitelistedTokens[_token].v2Router = _newRouter;
    }
```

This lack of equality check can result in redundant updates to the same address, causing unnecessary storage writes and event emissions.

Suggestion:

Add an equality check before the update to prevent setting the router to the same address as the current value:

```
if (whitelistedTokens[_token].v2Router == _newRouter)
    revert SameRouterNotAllowed();
```

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MST-3 `addModule` and `removeModule` Functions are Missing Event Emissions

Severity: Minor

Status: Fixed

Code Location:

src/MemeStrategy.sol#885 896

Descriptions:

In the `addModule` and `removeModule` functions, the contract modifies critical module whitelist states (`whitelistedModules` , `moduleList` , and `moduleIndex`) but does not emit any events. However, the event previously used for these operations was removed.

Suggestion:

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MST-4 Missing Existence Checks in `addMemToken` and `removeMemeToken`

Severity: Minor

Status: Fixed

Code Location:

src/MemeStrategy.sol#600-620

Descriptions:

Both `addMemeToken` and `removeMemeToken` functions lack proper existence checks when modifying the whitelist mapping.

- In `addMemeToken`, the function adds a token to the whitelist without verifying whether it is already whitelisted.
- In `removeMemeToken`, the function sets isWhitelisted = false without verifying that the token is currently whitelisted.

Suggestion:

Add explicit state validation before modifying whitelist entries.

```
function addMemeToken(address _token, address _v2Router) external
onlyOwnerOrOperator {
    if (_token == address(0) || _v2Router == address(0)) revert InvalidAddress();
    if (!approvedRouters[_v2Router]) revert RouterNotApproved();
    if (whitelistedTokens[_token].isWhitelisted) revert TokenAlreadyWhitelisted();

    whitelistedTokens[_token] = MemeToken({
        v2Router: _v2Router,
        isWhitelisted: true
    });

    emit MemeTokenWhitelisted(_token, _v2Router);
}
```

```
function removeMemeToken(address _token) external onlyOwnerOrOperator {
    if (!whitelistedTokens[_token].isWhitelisted) revert TokenNotWhitelisted();

    whitelistedTokens[_token].isWhitelisted = false;
    emit MemeTokenDelisted(_token);
}
```

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MST-5 Inconsistent Revert Message in onlyOwnerOrOperator Modifier

Severity: Informational

Status: Fixed

Code Location:

src/MemeStrategy.sol#340;

src/MemeStrategyHook.sol#292

Descriptions:

`MemeStrategy.sol` and `MemeStrategyHook.sol` each define a version of the `onlyOwnerOrOperator` modifier with identical logic: both check whether the caller (msg.sender) is the owner or an authorized operator.

```
modifier onlyOwnerOrOperator() {
    if (msg.sender != OWNER && !operators[msg.sender]) revert OnlyOwner();
    _;
}
```

```
modifier onlyOwnerOrOperator() {
    if (msg.sender != owner() && !operators[msg.sender])
        revert NotAuthorized();
    _;
}
```

However, they revert with different custom error messages — `OnlyOwner()` in one version and `NotAuthorized()` in the other — which could be misleading, especially the use of OnlyOwner(), as it does not indicate that operators are also permitted.

Suggestion:

Change the `onlyOwnerOrOperator` modifier in `MemeStrategyHook.sol` to use the same `NotAuthorized()` revert message as in `MemeStrategy.sol`.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MST-6 Comment Description Inconsistent with Code Implementation

Severity: Informational

Status: Fixed

Code Location:

src/MemeStrategy.sol#1238-1247

Descriptions:

The comment states `Current price must be >= targetEntryPrice (more tokens per ETH = better)` , but the code actually uses `pricePerToken > queue.targetEntryPrice` .

Suggestion:

Update the comment to correctly reflect the logic as: `Current price must be <= targetEntryPrice (more tokens per ETH = better).`

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MSV-1 Unused `purchaseId` Parameter Passed to `_addActivePurchase()` Function

Severity: Informational

Status: Fixed

Code Location:

src/MemeStrategyViews.sol#1634

Descriptions:

In the `_executeMemeTokenPurchase()` function, the protocol calls `_addActivePurchase()` to increase the active purchase counter and passes a parameter `purchaseId`. However, this `purchaseId` parameter is never used within the function.

```
function _addActivePurchase(uint256 /* _purchaseId */) internal {
    activePurchaseCount++;
}
```

Suggestion:

It is recommended to remove the unused `purchaseId` parameter from the `_addActivePurchase()` function to simplify the code and improve clarity.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.