RISK AUDIT

for

on

June 12, 2025

FIDESIUM

Executive Summary

Report

19
/100

TOTAL

Medium risk

Jun 12, 2025

Abstract

Fidesium's automated risk assessment service was requested to perform a risk posture audit on International Meme Fund **contracts**

Repository Link: https://github.com/International-Meme-Fund/markets-v2

Initial Commit Hash:

`d51b51c60f4358cf3acbfef201daf6c585216cf8`

Followup scan Commit Hash:

`4b9500451963da0c1a06c6d3894bccbbd89ac387`

Followup scan Commit Hash:

`a2000dbffc4e05ac5af7ba202ca38350224d8b76`

Issue Summary

Critical	High	Medium	Low	Info
1 Issues	4 3 Issues	~~7~~ 6 Issues	4 Issues	~~6~~ 7 Issues

Caveats

International Meme Fund's codebase is well written, but does incur a handful of high value flaws.

Test Approach

Fidesium performed both Whitebox and Blackbox testing, as per the scope of the engagement, and relied on automated security testing.

Methodology

The assessment methodology covered a range of phases and employed various tools, including but not limited to the following:

- Mapping Content and Functionality of API
- Application Logic Flaws
- Access Handling
- Authentication/Authorization Flaws
- Brute Force Attempt
- Input Handling
- Source Code Review
- Fuzzing of all input parameter
- Dependency Analysis

Severity Definitions

Critical	The issue can cause large economic losses, large-scale data disorder or loss of control of authority management.
High	The issue puts users' sensitive information at risk or is likely to lead to catastrophic financial implications.
Medium	The issue puts a subset of users' sensitive information at risk, reputation damage or moderate financial impact.
Low	The risk is relatively small and could not be exploited on a recurring basis, or is low-impact to the client's business.
Informational	The issue does not pose an immediate risk but is relevant to security best practices or defence in Depth.

Risk Issues

Vunerability	Description	Risk	Probability	Status
Presumption of standards compliant decimals	`UniV3Oracle` assumes `decimals` is standards compliant	Critical	Medium	Acknowledged
Presumption of succesful transfer	`safeTransfer` in `SafeTransferLib` presumes transfer was succesful when `returnData` is empty	High	Medium	Acknowledged
Reentrancy	The `deposit` and `withdraw` function make multiple external calls before updating state.	High	Medium	Resolved
Low TWAP Period	The `UniV3Oracle` has a very low TWAP period	High	Medium	Acknowledged
Low Liquidity allowed for TWAP	The `UniV3Oracle` does not validate liquidity	High	Medium	Acknowledged
Centralization	Multiple contracts rely on ~~`Ownable`~~ `OwnableUpgreadeable`	Medium	Medium	Acknowledged
Missing Access Control	`sync` function lacks access control	Medium	Medium	Active
One Step Ownership Transfer	Multiple contracts rely on ~~`Ownable`~~ `OwnableUpgreadeable`	Medium	Medium	Resolved
Missing Zero Address Validations	Multiple locations in the codebase are missing a zero address validation. This can result in unexpected behavior, and lost assets.	Medium	Medium	Remediated
Missing Oracle Validation	`whitelistTargetMarket` sets the oracle address without further validation	Medium	Medium	Active
MEV Sandwich Attacks: Missing Price Impact verification	Minimum amounts are set to 0.	Medium	Medium	Active
Missing contract validation	`ProxyOracle` does not valdate `delegate` address	Medium	Medium	Acknowledged
Reliance on Block Timestamp	Multiple functions rely on `block.timestamp`.	Medium	Unlikely	Active
Missing Bounds Validation	Multiple functions do not validate upper and/or lower bounds.	Medium	Low	Active
Presumption of approval success	The `IMFLiquidityManager` contract presumes success on `.approve` calls.	Low	Low	Remediated
Missing Uniswap fee tier validation	The `IMFLiquidityManager.whitelistTargetMarket` function does not validate fee validity.	Low	Low	Active
Missing immutable	`pool` in `DIAOracle` should be immutable.	Low	Low	Acknowledged
Reliance on Fixed Deadlines	Multiple functions rely on fixed deadlines.	Low	Unlikely	Active
Gas Ineffiency: Non Consolidated requires	`safeTransfer` in `SafeTransferLib` applies multiple requires on simultaneously available data	Info	Info	Active

Risk Issues

Vunerability	Description	Risk	Probability	Status
Gas Inefficiency: Redundant storage reads	`deposit` in `IMFLiquidityManager` has redundant storage reads	Info	Info	Active
Gas Inefficiency: Redundant storage reads	`deposit` in `IMFLiquidityManager` has redundant storage reads	Info	Info	Active
Gas Inefficiency: Redundant storage reads	`withdraw` in `IMFLiquidityManager` has redundant storage reads	Info	Info	Active
Gas Inefficiency: Redundant intermediate variable	`USDSPoolPercent` and `amountUSDSPoolMEMEDesired`in `IMFLiquidityManager` are redundant	Info	Info	Active
Gas Inefficiency: Unnecessary accumulation	`userMEMEAmount` in `IMFLiquidityManager._withdraw` is unnecessarily accumulated	Info	Info	Resolved
Gas Inefficiency: Duplicate casts	Multiple variables are cast repeatedly	Info	Info	Acknowledged
Gas Inefficiency: Long Revert Strings	`UniV3Oracle` has long revert strings	Info	Info	Acknowledged

Risk Overview

Team Risk

Low risk: 1

No issues found in founding team

Doxxing Status	Team Experience	Risk Summary
Public	Highly relevant	Low

Liquidity

Risk summary: N/A

As this is a Github assessment, liquidity risks have not been assessed

Whale Concentration

Risk summary: N/A

As this is a Github assessment, whale risks have not been assessed

Smart Contract Risks

Risk summary: 4826

The contracts are mostly well written, but have a handful of flaws that should to be carefuly managed.

Vulnerabilities Critical

Presumption of standards compliant decimals

Vulnerability severity: **Critical**

Vulnerability probability: **Medium**

`UniV3Oracle` assumes `decimals` is standards compliant

In the worst case a malicious developer could implement decimals causing gas exhaustion rendering the Oracle unusable

Recommendations:

```
uint8 public immutable baseTokenDecimals;
uint8 private constant MAX_REASONABLE_DECIMALS = 36;

function validateDecimals(address token) internal view returns (uint8) {
    (bool success, bytes memory data) = token.staticcall(
        abi.encodeWithSignature("decimals()")
    );

    require(success, "Decimals call failed");
    require(data.length == 32, "Invalid decimals return data");

    uint8 tokenDecimals = abi.decode(data, (uint8));

    require(tokenDecimals > 0, "Decimals cannot be zero");
    require(tokenDecimals <= MAX_REASONABLE_DECIMALS, "Decimals too large");

    return tokenDecimals;
}

constructor(address _pool, bool _baseAssetIsToken0, uint32 _period) {
    ...
    pool = IUniswapV3Pool(_pool);
    baseToken = _baseAssetIsToken0 ? pool.token0() : pool.token1();
    quoteToken = _baseAssetIsToken0 ? pool.token1() : pool.token0();

    baseTokenDecimals = validateDecimals(baseToken);

    baseTokenAmount = uint128(10 ** baseTokenDecimals);
    oracleScalar = 10 ** (36 - baseTokenDecimals);
    period = _period;
}
```

Vulnerabilities High

Low TWAP Period

Vulnerability severity: **High**

Vulnerability probability: **Medium**

The `UniV3Oracle` has a very low TWAP period

This is highly susceptible to manipulation

Recommendations:

Industry standard TWAP periods are considered to be 24 hours

Low Liquidity allowed for TWAP

Vulnerability severity: **High**

Vulnerability probability: **Medium**

The `UniV3Oracle` does not validate liquidity

This is highly susceptible to manipulation, Flash Loans, Sandwich Attacks, and Arbitrage

Recommendations:

Implement a minimum liquidity and volume requirement

Reentrancy

Vulnerability severity: **High**

Vulnerability probability: **Medium**

The `deposit` and `withdraw` function make multiple external calls before updating state.

Recommendations:

- Apply the `nonReentrant` modifier
- Ensure adherence to Checks-Effects-Interactions
- Move external calls after state updates

Action Taken:

Resolved at commit `6da42f31ce0cede381d20997826fe75d0c8943fa`

Vulnerabilities High

Presumption of succesful transfer

Vulnerability severity: **High**

Vulnerability probability: **Medium**

`safeTransfer` in `SafeTransferLib` presumes transfer was succesful when `returnData` is empty

An attacker could create and list a malicious token to manipulate the protocol, potentiall compounding impact by way of Flash Loans

Recommendations:

Add additional, explicit balance checks.

```
uint256 balanceAfter = token.balanceOf(address(this));
bool transferred = (balanceBefore - balanceAfter) == value;
require(transferred, ErrorsLib.TRANSFER_BALANCE_VERIFICATION_FAILED);
```

Vulnerabilities Medium

Centralization

Vulnerability severity: **Medium**

Vulnerability probability: **Medium**

Multiple contracts rely on `Ownable`

Recommendations:

- Introduce more fine grained access controls
- Ensure `owner` is a well managed multisig

Missing Access Control

Vulnerability severity: **Medium**

Vulnerability probability: **Medium**

`sync` function lacks access control

This opens the protocol up to market manipulation, MEV exploitation, dillution attacks, and gas wars.

During extreme market conditions this could result in a DoS.

Recommendations:

There are multiple ways to secure this function, each with their own tradeoffs

1. Limit sync to be callable only by Owner/Admin
2. Implement a Timelock on sync operations
3. Governance control
4. Require a staked bond before calling sync, and return only on improved protocol incentives

If centralization is not a primary concern, option (1) is the easiest and cleanest solution. Else some combination of the other three is recommended.

Vulnerabilities Medium

Missing Zero Address Validations

Vulnerability severity: **Medium**

Vulnerability probability: **Medium**

Multiple setters in the codebase are missing a zero address validation. This can result in unexpected behavior, and lost assets.

Contract	Function	Parameter
IMFLiquidityManager	constructor	morphoVaultAddress
IMFLiquidityManager	withdraw	marketAddress
IMFLiquidityManager	freezeMarket	marketAddress
IMFLiquidityManager	unfreezeMarket	marketAddress
IMFLiquidityManager	_mintPosition	token0
IMFLiquidityManager	_mintPosition	token1
IMFLiquidityManager	_decreaseAndCollectUSDSMEME	USDSMEMEPool
ProxyOracle	constructor	delegate
TwoHopOracle	constructor	oracle1
TwoHopOracle	constructor	oracle2

Recommendations:

Use `!= address(0)` to validate these parameters are not zero addresses

Action Taken:

Partially Remediated

Vulnerabilities Medium

Missing contract validation

Vulnerability severity: **Medium**

Vulnerability probability: **Medium**

`ProxyOracle` does not valdate `delegate` address

Recommendations:

Ensure `delegate` is a valid, non EOA contract and conforms to expected ABI

One Step Ownership Transfer

Vulnerability severity: **Medium**

Vulnerability probability: **Medium**

Multiple contracts rely on `OwnableUpgreadeable`

Ownership transfer is a single step operation, which could lead to loss of protocol control

Recommendations:

Rely on `Ownable2StepUpgradeable`

Action Taken:

Resolved at commit `8ce7e18fe69b1f69e27a3461a1b90c0bd8dc5c35`

Vulnerabilities Medium

Missing Oracle Validation

Vulnerability severity: **Medium**

Vulnerability probability: **Medium**

`whitelistTargetMarket` sets the oracle address without further validation

```
...
...
markets[marketAddress].oracleAddress = oracleAddress;
...
...
```

There are no validations that the address conforms to an expected interface, or its functionality or data quality.

Recommendations:

Validate the oracle implementation

```
mapping(address => bool) public approvedOracleImplementations;

function isApprovedOracleImplementation(address oracleAddress) internal view returns (bool) {
    bytes32 codeHash;
    assembly {
        codeHash := extcodehash(oracleAddress)
    }

    return approvedOracleImplementations[oracleAddress] ||
           approvedOracleCodeHashes[codeHash];
}

try IOracle(oracleAddress).isValid() returns (bool isValid) {
    require(isValid, ErrorsLib.ORACLE_NOT_VALID);
} catch {
    revert(ErrorsLib.INVALID_ORACLE_INTERFACE);
}

try IOracle(oracleAddress).getPrice(marketAddress, address(USDS)) returns (uint256 price, uint256 timestamp) {
    require(price > 0, ErrorsLib.ZERO_PRICE);
    require(block.timestamp - timestamp < 1 hours, ErrorsLib.STALE_ORACLE_DATA);
} catch {
    revert(ErrorsLib.ORACLE_FETCH_FAILED);
}

require(
  isApprovedOracleImplementation(oracleAddress),
  ErrorsLib.UNAPPROVED_ORACLE_IMPLEMENTATION
);
```

Vulnerabilities Medium

MEV Sandwich Attacks: Missing Price Impact verification

Vulnerability severity: **Medium**

Vulnerability probability: **Medium**

Minimum amounts are set to 0

```
...
params.amount0Min = 0;
params.amount1Min = 0;
...
decreaseParams.amount0Min = 0;
decreaseParams.amount1Min = 0;
...
decParams.amount0Min = 0;
decParams.amount1Min = 0;
...
swapParamsIn.amountOutMinimum = 0;
...
```

Recommendations:

- Implement slippage protection
- Implement max input checks
- Validate Price Impact
- Implement bounds for withdrawal/swap

```
(uint160 sqrtPriceX96, , , , , , ) = IUniswapV3Pool(pool).slot0();
...
(int24 twapTick, ) = OracleLibrary.consult(pool, TWAP_INTERVAL);
uint160 twapSqrtPriceX96 = TickMath.getSqrtRatioAtTick(twapTick);
...
uint160 usedSqrtPriceX96 = sqrtPriceX96 < twapSqrtPriceX96 ?
    sqrtPriceX96 : twapSqrtPriceX96;
...
uint256 expectedOut = UniswapV3Library.getQuoteAtSqrtRatio(
   usedSqrtPriceX96,
   amountIn,
   tokenIn,
   tokenOut
);
minAmountOut = expectedOut - ((expectedOut * MAX_SLIPPAGE_BP) / 10000);
```

- Implement TWAP checks
- Implement Flashbots integration for MEV protection

Vulnerabilities Medium

Missing Bounds Validation

Vulnerability severity: **Medium**

Vulnerability probability: **Low**

Multiple functions do not validate upper and/or lower bounds.

Contract	Function	Parameter	Validation
IMFLiquidityManager	deposit	amount	Upper

Recommendations:

Validate bounds

Reliance on Block Timestamp

Vulnerability severity: **Medium**

Vulnerability probability: **Low**

Multiple functions rely on `block.timestamp`, which can be manipulated by miners.

Recommendations:

- Use block numbers instead of timestamps.
- If timestamps are necessary, use trusted external oracles.

Vulnerabilities Low

Missing Uniswap fee tier validation

Vulnerability severity: **Low**

Vulnerability probability: **Low**

The `IMFLiquidityManager.whitelistTargetMarket` function does not validate fee validity.

Uniswap v3 allows specific feeTiers. While the function succesfully reverts on nonexistent pools, this could still lead to wasted gas

Recommendations:

Explicityly validate pool fee validity.

```
mapping(uint24 => bool) private validFeeTiers;

validFeeTiers[100] = true;
validFeeTiers[500] = true;
validFeeTiers[3000] = true;
validFeeTiers[10000] = true;

function _isValidFeeTier(uint24 fee) internal view returns (bool) {
    return validFeeTiers[fee];
}

}

require(_isValidFeeTier(poolFee), "Invalid Uniswap fee tier");
```

Vulnerabilities Low

Presumption of approval success

Vulnerability severity: **Low**

Vulnerability probability: **Low**

The `IMFLiquidityManager` contract presumes success on `.approve` calls.

```
MEME.approve(address(v3PositionManager), amountUSDSPoolMEMEDesired);
```

A malicious contract could selectively fail approvals and disrupt protocol operations

Gas could be wasted due to non standards compliant approval implementations

Recommendations:

Implement a safeApprove function

```
function safeApprove(IERC20 token, address spender, uint256 value) internal {
    require(address(token).code.length > 0, ErrorsLib.NO_CODE);

    if (value > 0) {
        (bool resetSuccess, bytes memory resetReturndata) =
            address(token).call(abi.encodeCall(IERC20Internal.approve, (spender, 0)));
        if (!resetSuccess || (resetReturndata.length != 0 && !abi.decode(resetReturndata, (bool)))) {
            revert(resetSuccess ? ErrorsLib.APPROVE_RETURNED_FALSE : ErrorsLib.APPROVE_REVERTED);
        }
    }

    (bool success, bytes memory returndata) =
        address(token).call(abi.encodeCall(IERC20Internal.approve, (spender, value)));
    if (!success || (returndata.length != 0 && !abi.decode(returndata, (bool)))) {
        revert(success ? ErrorsLib.APPROVE_RETURNED_FALSE : ErrorsLib.APPROVE_REVERTED);
    }
}
```

Action Taken:

Remediated at commit `013b1d5ae6994a00d9dcb37755bf7b5eda15781f`

Missing immutable

Vulnerability severity: **Low**

Vulnerability probability: **Low**

`pool` in `DIAOracle` should be immutable.

`pool` has no setters but is not `immutable`

Recommendations:

Set `pool` as `immutable`

Vulnerabilities Low

Reliance on Fixed Deadlines

Vulnerability severity: **Low**

Vulnerability probability: **Low**

Multiple functions rely on fixed deadlines

```
params.deadline = block.timestamp + 1 hours;
```

Network congestion could delay transactions beyond this window, and MEV bots can manipulate transaction ordering within this window

Additionally, since this deadline is set at runtime, mempool exposure increases the MEV bot risk

Recommendations:

- Monitor for high gas periods/network congestion, and dynamically adjust execution window based on gas cost
- Allow for configurable, preferably user configurable deadlines
- Ensure slippage protection is applied

Vulnerabilities Info

Gas Inefficiency: Redundant intermediate variable

Vulnerability severity: **Info**

Vulnerability probability: **Info**

`USDSPoolPercent` and `amountUSDSPoolMEMEDesired`in `IMFLiquidityManager` are redundant

`USDSPoolPercent` is set to 100, but only used `uint256 amountUSDSPoolMEMEDesired = (amount * USDSPoolPercent) / 100;`

Recommendations:

Remove `amountUSDSPoolMEMEDesired` and `USDSPoolPercent`

Action Taken:

Resolved at commit `013b1d5ae6994a00d9dcb37755bf7b5eda15781f`

Gas Inefficiency: Unnecessary accumulation

Vulnerability severity: **Info**

Vulnerability probability: **Info**

`userMEMEAmount` in `IMFLiquidityManager._withdraw` is unnecessarily accumulated

```
userMEMEAmount = 0;
userMEMEAmount += (specificUserPosition.liquidityShares * MEMEPoolAmount) / market.totalStakedShares;
```

Recommendations:

Set `userMEMEAmount` directly

```
userMEMEAmount = (specificUserPosition.liquidityShares * MEMEPoolAmount) / market.totalStakedShares;
```

Action Taken:

Resolved at commit `013b1d5ae6994a00d9dcb37755bf7b5eda15781f`

Vulnerabilities Info

Gas Ineffiency: Non Consolidated requires

Vulnerability severity: **Info**

Vulnerability probability: **Info**

`safeTransfer` in `SafeTransferLib` applies multiple requires on simultaneously available data

Recommendations:

Collapse the `require` calls into a single check and rely on `revert`. This will save ~ 200 gas per call.

```
if (!success || (returndata.length != 0 && !abi.decode(returndata, (bool)))) {
    revert(success ? ErrorsLib.TRANSFER_RETURNED_FALSE : ErrorsLib.TRANSFER_REVERTED);
}
```

Vulnerabilities Info

Gas Inefficiency: Duplicate casts

Vulnerability severity: **Info**

Vulnerability probability: **Info**

Multiple variables are cast repeatedly

- address(v3PositionManager)
- address(USDS)
- address(token)
- address(this)

Recommendations:

Ensure casts only happen once, before usage

Gas Inefficiency: Long Revert Strings

Vulnerability severity: **Info**

Vulnerability probability: **Info**

`UniV3Oracle` has long revert strings

Revert strings above 32 bytes consume significantly more gas

Recommendations:

Shorten revert strings to fit in 32 bytes

Vulnerabilities Info

Gas Inefficiency: Redundant storage read

Vulnerability severity: **Info**

Vulnerability probability: **Info**

`deposit` in `IMFLiquidityManager` has redundant storage reads

Each read of `specificUserPosition` requires a storage read

Recommendations:

Create a memory struct to avoid redundant `SLOAD`

```
UserMarketPosition storage userMarketPositions = users[msg.sender][marketAddress];
userMarketPositions.positions.push();
UserPositions storage specificUserPosition = userMarketPositions.positions[userIndex];

uint256 liquidityShares;
if (market.totalStakedShares == 0) {
    liquidityShares = amount;
} else {
    liquidityShares = (amount * market.totalStakedShares) / MEMEPoolAmount;
}

specificUserPosition.depositAmountMEME = amount;
specificUserPosition.liquidityShares = liquidityShares;
```

Gas Inefficiency: Redundant storage read

Vulnerability severity: **Info**

Vulnerability probability: **Info**

`deposit` in `IMFLiquidityManager` has redundant storage reads

Each read of `market.*` requires a storage read

Recommendations:

Cache market values to avoid redundant storage reads

```
...
...
uint256 currentNftIndex = market.USDSPoolMEMESide.nftIndex;
uint128 currentLiquidity = market.USDSPoolMEMESide.liquidity;
address poolAddress = market.USDSPoolAddress;
uint24 poolFee = market.USDSPoolFee;
uint256 currentTotalStakedShares = market.totalStakedShares;
...
...
```

Vulnerabilities Info

Gas Inefficiency: Redundant storage read

Vulnerability severity: **Info**

Vulnerability probability: **Info**

`withdraw` in `IMFLiquidityManager` has redundant storage reads

Each read of `market.*` requires a storage read

Recommendations:

Cache market values to avoid redundant storage reads, and conduct all computation on cached values

```
...
...
Market storage market = markets[marketAddress];
uint256 totalSharesInMemory = market.totalStakedShares;
uint256 morphoTPSInMemory = market.totalMORPHOTPS;
...
...
totalSharesInMemory += newShares;
morphoTPSInMemory = (morphoTPSInMemory * oldValue + newValue) / totalSharesInMemory;
...
...
market.totalStakedShares = totalSharesInMemory;
market.totalMORPHOTPS = morphoTPSInMemory;
```

Disclaimer

Disclaimer

This report is governed by the Fidesium terms and conditions.

This report does not constitute an endorsement or disapproval of any project or team, nor does it reflect the economic value or potential of any related product or asset. It is not investment advice and should not be used as the basis for investment decisions. Instead, this report provides an assessment intended to improve code quality and mitigate risks inherent in cryptographic tokens and blockchain technology.

Fidesium does not guarantee the absence of bugs or vulnerabilities in the technology assessed, nor does it comment on the business practices, models, or regulatory compliance of its creators. All services, reports, and materials are provided "as is" and "as available," without warranties of any kind, including but not limited to merchantability, fitness for a particular purpose, or non-infringement.

Cryptographic assets and blockchain technologies are novel and carry inherent technical risks, uncertainties, and the possibility of unpredictable outcomes. Assessment results may contain inaccuracies or depend on third-party systems, and reliance on them is solely at the Customer’s risk.

Fidesium assumes no liability for content inaccuracies, personal injuries, property damages, or losses related to the use of its services, reports, or materials. Third-party components are provided "as is," and any warranties are strictly between the Customer and the third-party provider.

These services and materials are intended solely for the Customer's use and benefit. No third party or their representatives may claim rights to or rely on these services, reports, or materials under any circumstances.