IRONNODE

ironnode.io

Memeswap.fi

EVM Smart Contract Security Audit

by **IronNode** from **May 1 - May 28, 2024**

1	Executive overview	3
1.1	Introduction	4
1.2	Audit summary	4
	Objectives of the audit	5
1.3	Scope	5
1.4	Assessment summary & findings overview	6
2	Application Flow Analysis	7
2.1	Token Creator Role	8
	Description	8
	Functions	8
2.2	Rent Provider Role	9
	Description	9
	Functions	9
2.3	Admin Role	9
	Description	9
	Functions	9
4	Findings	11
4.1	Re-Entrancy in Rent Function	12
	Code location	12
	Recommendation	13

4.2 Access Control Issue for rent Function 14

Code location 14

Recommendation 16

4.3 Possible Queue Spamming in Dequeue Process 15

Code location 15

Recommendation 15

4.4 Unbounded Gas Loops 16

Code location 16

Recommendation 16

4.5 Unbounded Service Fee Setting 17

Code location 17

Recommendation 17

4.6 Unused FlashLoan Functionality 18

Recommendation 18

5 Manual testing 20 - 22

Executive overview

1 Executive overview

1.1 Introduction

Memeswap.fi is a decentralized platform designed to streamline token launches with ease and efficiency. By allowing projects to deploy tokens with customized tax options and rent ETH as pair liquidity, Memeswap enables instant and effortless token launches for a minimal fee. The platform prioritizes trader security with pre-defined token contracts, ensuring a transparent and safe trading environment, enhanced by UI signals and push notifications.

Memeswap also introduces time-locked LPs, mandating a minimum one-day lock on initial liquidity to protect traders. Innovative rented liquidity features allow projects to rent ETH from Staking Vaults, eliminating the need to raise large sums upfront. This system offers dynamic pricing and incentives for stakers, ensuring constant liquidity availability. Additionally, Memeswap provides secure real yield opportunities for ETH stakers, with rentals capped at the total token supply to prevent financial losses, creating a reliable and secure ecosystem for both traders and project developers.

In response to a request from Memeswap.fi, IronNode, conducted a thorough security audit of the Memeswap.fi EVM protocol to ensure its integrity, security, and user trustworthiness. The audit was carried out from May 1st, 2024 to May 28th, 2024.

1.2 Audit summary

The security audit team at IronNode was allocated three weeks to conduct an extensive review of the Memeswap.fi protocol. A dedicated team, including a lead security engineer with profound expertise in blockchain technology, smart contract security, and cybersecurity, spearheaded the audit.

1.3 Findings and Remediations

During the audit process of the EVM smart contracts for the Memeswap protocol, a total of six issues were identified, categorized as follows: one high-severity issue, four low-severity issues, and one informational issue. Each of these issues was thoroughly documented and communicated to the Memeswap development team for remediation.

In response, the Memeswap team promptly addressed all the identified issues. They undertook a comprehensive review and implemented necessary code changes to rectify each concern. The resolution of these issues was confirmed in their commit with the hash: f5e44d6e57e352266cbeb6a847812e12dc1dd78b.

The successful remediation of these issues underscores Memeswap's commitment to maintaining a secure and reliable platform for its users. Their proactive and thorough approach in addressing the audit findings ensures a higher level of trust and confidence in the protocol's ongoing operations.

1.2.1 Objectives of the audit

- **1. Smart Contract Security**
  - **Code Review:** Identify vulnerabilities and enforce best practices.
  - **Re-entrancy Protection:** Verify the use of re-entrancy guards.
  - **Access Controls:** Ensure proper authorization for critical functions.
  - **Data Integrity:** Prevent data manipulation vulnerabilities.
  - **Arithmetic Errors:** Protect against integer overflow/underflow issues.
  - **Contract Upgradability**: Secure and reliable upgrade mechanisms.

2. Token Launch Mechanisms

- **Tax Customization:** Validate customizable tax options on tokens.
- **Liquidity Rental:** Assess security and efficiency of ETH rental mechanisms.
- **Deployment Process:** Ensure a secure and user-friendly token deployment.

3. Trader Security

- **Pre-defined Contracts:** Ensure contracts prevent unexpected behaviors.
- **UI Signals and Notifications:** Enhance security and user experience.
- **Time-Locked LPs:** Enforce a minimum lock period for initial liquidity.

4. Staking and Liquidity Provision

- **Staking Mechanisms:** Secure and verify ETH staking processes.
- **Liquidity Rental Security:** Ensure safe dynamic pricing and incentives.
- **Fair Launch Conditions:** Prevent over-rental and financial losses.

5. Yield Security

- **Real Yield Security:** Ensure no financial losses for ETH stakers.
- **Incentive Mechanisms:** Provide fair and secure staking incentives.
- **Yield Calculation:** Guarantee accurate and secure reward distribution.

1.3 Scope

1.3.1 Code repositories

memeswap-contracts

Branch: main

Commit ID: https://github.com/tkzo/memeswap-contracts/commit/092a9d6f9a9cc918afd834d466ffa6cdc47b0568

1.4 Assessment summary & findings overview

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	0	4	1

Security analysis	Risk Level	Remediation Status
Re-Entrancy in Rent Function	HIGH	FIXED
Access Control Issue for rent Function	LOW	FIXED
Possible Queue Spamming in Dequeue Process	LOW	FIXED
Unbounded Gas Loops	LOW	FIXED
Unbounded Service Fee Setting	LOW	FIXED
Unused FlashLoan Functionality	INFORMATIONAL	FIXED

Application Flow Analysis

2 MemeswapFi Protocol

This section of the report provides an in-depth analysis of the application flow for the MemeswapFi Protocol contract. The analysis covers the logical flow of the code, with an emphasis on two primary roles: Token Creator, Trader and Admin.

In the Memeswap protocol, “rent” allows Token Creators to borrow ETH from the MemeswapVault to provide initial liquidity for their tokens. During the token launch, the launch function in the MemeswapTokenFactory contract calculates the required tokens, converts ETH to WETH, and adds liquidity to the token-ETH pair. If the token meets certain criteria, such as maintaining sufficient liquidity or duration, it reaches “Valhalla,” where the token supply is returned to the creator, rewarding them for achieving stability and growth.

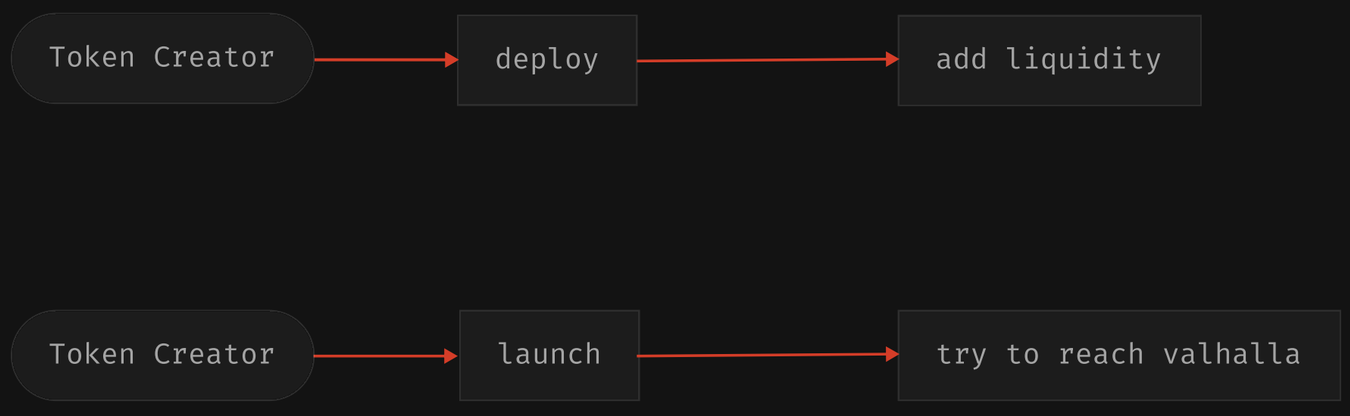
2.1 Token Creator Role

The Token Creator role involves deploying new tokens and managing their initial liquidity on the Memeswap platform. Token Creators can deploy a token using the deploy function, add liquidity to the token using the addLiquidity function, and launch the token with rented ETH using the launch function. If the token meets certain criteria, it reaches “Valhalla,” where the token supply is returned to the creator as a reward.

2.1.1 Functions

- 1. deploy: Allows users to create a new token with specified parameters.
  - Explanation: Deploys a new token with given details.
  - Contract: MemeswapTokenFactory
- 2. addLiquidity: Allows users to add liquidity to a token pair.
  - Explanation: Adds liquidity to the token-ETH pair.
  - Contract: MemeswapRouter
- 3. launch: Allows users to launch a new token with initial liquidity provided by renting ETH.
  - Explanation: Launches a token and rents ETH for initial liquidity.
  - Contract: MemeswapTokenFactory
- 4. Reaching Valhalla: Automatically returns the token supply to the creator if certain criteria are met.
  - Explanation: Returns token supply to the creator when criteria are met.
  - Contract: MemeswapVault

2.1.2 Flow



2.2 Rent Provider Role

The Rent Provider role involves staking ETH into the MemeswapVault to earn yield and providing liquidity for new tokens by lending ETH. Rent Providers can queue their staked ETH for rent, earn rewards from the staking process, and manage their liquidity provision.

2.2.1 Functions

- 1. stake: Allows users to stake ETH into the vault.
  - Explanation: Stakes ETH into the MemeswapVault to earn yield.
  - Contract: MemeswapVault
- 2. enqueue: Allows users to queue their staked ETH for withdrawal.
  - Explanation: Queues staked ETH for withdrawal.
  - Contract: MemeswapVault
- 3. claimRewards: Allows users to claim earned rewards from staking.
  - Explanation: Claims rewards earned from staking ETH.
  - Contract: MemeswapVault



2.3 Admin Role

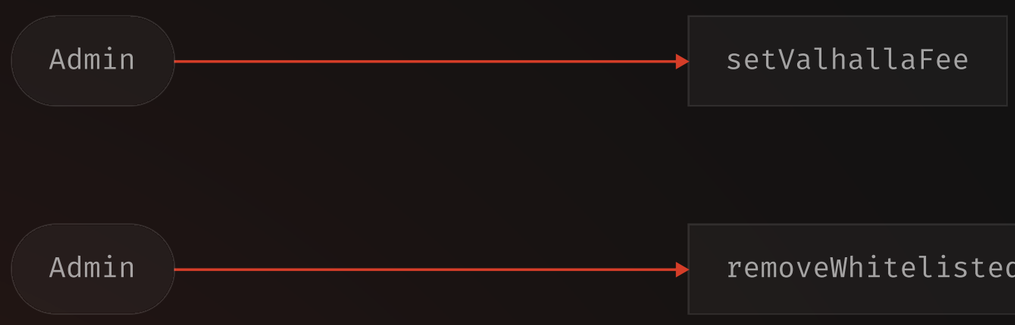
The Admin role involves managing the Memeswap protocol by setting parameters, handling token whitelisting, managing durations, monitoring, and responding to emergency situations.

2.2.1 Functions

- 1. setParams: Allows the admin to set parameters such as maxAPR, minAPR, fee, and duration.
  - Explanation: Sets various parameters for the vault operations.
  - Contract: MemeswapVault
- 2. setChadBar: Allows the admin to set the threshold for the Chad Bar.
  - Explanation: Sets the Chad Bar threshold for liquidity conditions.
  - Contract: MemeswapVault
- 3. setValhallaFee: Allows the admin to set the Valhalla fee.
  - Explanation: Sets the fee for processing tokens reaching Valhalla.
  - Contract: MemeswapVault
- 4. removeWhitelistedToken: Allows the admin to remove a token from the whitelist.
  - Explanation: Removes a token from the whitelist.
  - Contract: MemeswapFactory
- 5. isAllowedDuration: Checks if a duration is allowed.
  - Explanation: Verifies if a specific duration is permitted.
  - Contract: MemeswapFactory







Findings

4 Findings

4.1 Re-Entrancy In Rent

The MemeswapVault contract's rent function, when combined with the trigger modifier and the dequeue function, is vulnerable to re-entrancy attacks. The dequeue function sends ETH back to a queued user via a low-level call, allowing a malicious contract to re-enter the rent function and cause the dequeue function to be called multiple times.

4.1.1 Code location

rent Function:

```
    function rent(
        address _pair,
        address _token,
        uint256 _amount,
        uint256 _duration,
        address _user,
        address _pairToUnlock
    ) external trigger(_pairToUnlock) onlyOwner {
        if (!IMemeswapTokenFactory(owner()).isMemeswapToken(_token))
            revert FORBIDDEN();
        if (_amount == 0) revert ZERO_AMOUNT();
        if (!canRent(_amount)) revert VAULT_DRY();
        if (!IMemeswapFactory(factory).isAllowedDuration(_duration))
            revert WRONG_DURATION();
        uint256 price = getPriceForAmount(_amount, _duration);
        uint256 cut = takeCut(price);
        if (price - cut < duration) revert WRONG_AMOUNT();
        notifyRewardAmount(price - cut);
        IERC20(weth).transferFrom(address(this), msg.sender, _amount);
        rents[_pair] = Rent(
            _user,
            _token,
            _amount,
            _duration,
            block.timestamp + _duration
        );
        rentedSupply += _amount;
        emit RentalCreated(_pair, _amount, _duration);
    }
```

dequeue Function:

```
    function dequeue() private updateReward(queue[queueFirst].user) {
        uint256 amount = queue[queueFirst].amount;
        address user = queue[queueFirst].user;
        IWETH(weth).withdraw(amount);
        (bool success, ) = queue[queueFirst].user.call{value: amount}("");
        if (!success) revert TRANSACTION_FAILED();
        queueFirst++;
        userTotalQueue[user] -= amount;
        totalInQueue -= amount;
        totalSupply -= amount;
        balances[user] -= amount;
        emit Withdrawn(user, amount);
    }
```

4.1.2 Recommendation

To mitigate this issue, it is recommended tı use OpenZeppelin's ReentrancyGuard to add a nonReentrant modifier to the rent function. Additionally, follow the check-effects-interactions pattern to ensure state changes occur before any external interactions.

Example: Apply nonReentrant Modifier to Rent Function

```
    function rent(
        // existing code
    ) external trigger(_pairToUnlock) onlyOwner nonReentrant {
        // existing code
    }
```

Example: Use Temporary Variables

```
    function dequeue() private updateReward(queue[queueFirst].user) {
        uint256 amount = queue[queueFirst].amount;
        address user = queue[queueFirst].user;

        // Use temporary variables
        uint256 newQueueFirst = queueFirst + 1;
        uint256 newUserTotalQueue = userTotalQueue[user] - amount;
        uint256 newTotalInQueue = totalInQueue - amount;
        uint256 newTotalSupply = totalSupply - amount;
        uint256 newBalance = balances[user] - amount;

        // State changes
        queueFirst = newQueueFirst;
        userTotalQueue[user] = newUserTotalQueue;
        totalInQueue = newTotalInQueue;
        totalSupply = newTotalSupply;
        balances[user] = newBalance;

        // External interaction last
        IWETH(weth).withdraw(amount);
        (bool success, ) = user.call{value: amount}("");
        if (!success) revert TRANSACTION_FAILED();
```

4.2 Access Control Issue For Rent Function

The rent function in the MemeswapVault contract lacks proper access control, allowing any external address to call the function. This functionality should be restricted such that only the MemeswapTokenFactory contract can call the rent function to ensure controlled and authorized renting of liquidity.

This vulnerability allows unauthorized addresses to call the rent function, potentially leading to misuse of the liquidity renting mechanism, and disruption of the intended workflow of the platform.

4.2.1 Code location

rent Function:

```
function rent(
    address _pair,
    address _token,
    uint256 _amount,
    uint256 _duration,
    address _user,
    address _pairToUnlock
) external trigger(_pairToUnlock) {
//... Function Logic ...
    rents[_pair] = Rent(
        _user,
        _token,
        _amount,
        _duration,
        block.timestamp + _duration
    );
    rentedSupply += _amount;
    emit RentalCreated(_pair, _amount, _duration);
```

4.2.2 Recommendation

To mitigate this issue, it is recommended to implement access control to restrict the rent function such that only the MemeswapTokenFactory contract can call it. This can be achieved by adding a modifier that checks if the caller is the MemeswapTokenFactory contract.

```
modifier onlyTokenFactory() {
    require(msg.sender == address(IMemeswapTokenFactory(owner())),
      "Only TokenFactory can call this function");
    _;
}
```

4.3 Possible Queue Spamming In Dequeue Process

The enqueue and dequeue functionalities in the MemeswapVault contract allow users to add and remove their stakes from the queue. However, there are two primary concerns:

An attacker can spam the enqueue function with numerous small transactions, filling the queue and making it difficult for legitimate users to enqueue their withdrawals.

4.3.1 Code location

enqueue Function:

```
    function enqueue(uint256 _amount, address _pair) external trigger(_pair) {
        if (_amount == 0) revert ZERO_AMOUNT();
        if (_amount > balances[msg.sender] + userTotalQueue[msg.sender])
            revert WRONG_AMOUNT();
        queue[queueLast] = Queue(msg.sender, _amount);
        queueLast++;
        userTotalQueue[msg.sender] += _amount;
        totalInQueue += _amount;
        emit Enqueued(msg.sender, _amount);
    }
```

dequeue Function:

```
    function dequeue() private updateReward(queue[queueFirst].user) {
        uint256 amount = queue[queueFirst].amount;
        address user = queue[queueFirst].user;
        IWETH(weth).withdraw(amount);
        (bool success, ) = queue[queueFirst].user.call{value: amount}("");
        if (!success) revert TRANSACTION_FAILED();
        queueFirst++;
        userTotalQueue[user] -= amount;
        totalInQueue -= amount;
        totalSupply -= amount;
        balances[user] -= amount;
        emit Withdrawn(user, amount);
    }
```

4.3.2 Recommendation

To mitigate this issue, it is recommended to implement a minimum stake requirement and a maximum queue size, making it harder for attackers to spam the queue with small transactions and ensuring the dequeue process remains efficient.

4.4 Unbounded Gas Loops

The removeWhitelistedToken and isAllowedDuration functions in the MemeswapFactory contract contain loops that iterate over arrays (`whitelistedTokens` and `allowedDurations`). If these arrays grow large, the gas cost of executing these functions can become prohibitively high, leading to potential denial of service (DoS) vulnerabilities. This can make it impossible to remove tokens from the whitelist or check allowed durations if the arrays are too large.

4.4.1 Code location

removeWhitelistedToken Function:

```
function removeWhitelistedToken(address token) external onlyOwner {
    for (uint256 i = 0; i < whitelistedTokens.length; i++) {
        if (whitelistedTokens[i] == token) {
            whitelistedTokens[i] = whitelistedTokens[whitelistedTokens.length - 1];
            whitelistedTokens.pop();
            break;
```

isAllowedDuration Function:

```
function isAllowedDuration(uint256 _duration) external view returns (bool) {
    for (uint256 i = 0; i < allowedDurations.length; i++) {
        if (allowedDurations[i] == _duration) {
            return true;
        }
    }
    return false;
}
```

4.4.2 Recommendation

To mitigate the potential denial of service (DoS) vulnerabilities, it is recommended to add optional parameters start_index and end_index to limit the range of the array being searched. If these parameters are not provided, the functions will iterate over the entire array.

4.5 Unbounded Service Fee Setting

The setServiceFee function in the MemeswapFactory contract allows the contract owner to set the service fee to any arbitrary value. This could lead to potential abuse where the service fee is set to an extremely high value, negatively impacting the users and the overall functionality of the platform.

4.5.1 Code location

setServiceFee Function:

```
function setServiceFee(uint256 _serviceFee) external onlyOwner {
    serviceFee = _serviceFee;
}
```

4.5.2 Recommendation

To mitigate this vulnerability, it is recommended to set an upper limit should be defined for the service fee. This limit ensures that the service fee remains within a reasonable range, preventing potential abuse.

4.6 Unused FlashLoan Functionality

The MemeswapPair.sol contract contains a FlashLoan functionality within the swap function, called via memeswapV2Call. Although this functionality is not currently used anywhere in the application, it still poses a risk. FlashLoans can be exploited for economic attacks or other disruptive actions, even if they do not present an immediate security risk.

4.6.1 Code location

contracts/oracle_helper/src/contract.rs (Lines 50-61)

```
function swap(uint256 amount0Out, uint256 amount1Out, address to,
 bytes calldata data) external lock {
    require(amount0Out > 0 || amount1Out > 0, 'Memeswap: INSUFFICIENT_OUTPUT_AMOUNT');
//...Function code...

        if (data.length > 0) IMemeswapCallee(to).memeswapV2Call(msg.sender, amount0Out,
         amount1Out, data);  // FlashLoan call

        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
    }

 //... Function code...

    _update(balance0, balance1, _reserve0, _reserve1);
    emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
}
```

4.6.2 Recommendation

To mitigate the risk associated with this unused FlashLoan functionality, it is recommended to remove the memeswapV2Call invocation from the swap function. This will eliminate the potential for future exploitation through economic attacks or other disruptions.

Manual testing

5 Manual testing

5.1 MemeswapCollector.sol

- **1. Function: collectFees**
  - Validate that only authorized users can call collectFees.
  - Ensure proper calculation and transfer of fees.
  - Test for reentrancy attacks.
  - Check for integer overflow/underflow during fee calculations.

5.2 MemeswapFactory.sol

- **1. Function: createPair**
  - Validate that it is not possible to create a pair with the same token.
  - Ensure pairs can only be created once for the same token combination.
  - Check for proper initialization of pair contracts.
  - Verify that only authorized users can create pairs.
  - Test for reentrancy vulnerabilities during pair creation.
  - Ensure the correct emission of events during pair creation.

- **2. Function: removeWhitelistedToken**
  - Validate efficient handling of large whitelistedTokens arrays using start_index and end_index.
  - Ensure proper updating of the array and index mappings.
  - Check for edge cases such as removing the first and last tokens in the array.
  - Validate that only authorized users can remove whitelisted tokens.

- **3. Function: isAllowedDuration**
  - Validate correct iteration using start_index and end_index.
  - Ensure the function returns correct results for allowed and disallowed durations.
  - Test with boundary values of the duration array.
  - Ensure efficient performance with large duration arrays.

5.3 MemeswapLock.sol

- **1. Function: lockTokens**
  - Validate that tokens are correctly locked for the specified duration.
  - Ensure proper handling of already locked tokens.
  - Check for reentrancy vulnerabilities.
  - Verify that only authorized users can lock tokens.

- **2. Function: unlockTokens**
  - Validate correct unlocking of tokens after the lock duration.
  - Ensure proper handling of early unlock attempts.
  - Test for reentrancy vulnerabilities.
  - Ensure the correct emission of events during unlocking.

- **3. Function: extendLockDuration**
  - Validate correct extension of lock duration.
  - Ensure proper handling of invalid extension attempts.
  - Check for reentrancy vulnerabilities.
  - Verify that only authorized users can extend lock durations.

5.4 MemeswapPair.sol

- **1. Function: addLiquidity**
  - Validate correct addition of liquidity to the pair.
  - Ensure proper handling of edge cases (e.g., zero liquidity).
  - Check for reentrancy vulnerabilities.
  - Verify the correct calculation of liquidity tokens.
- **2. Function: removeLiquidity**
  - Validate correct removal of liquidity from the pair.
  - Ensure proper handling of edge cases (e.g., removing all liquidity).
  - Check for reentrancy vulnerabilities.
  - Verify the correct calculation of withdrawn amounts.
- **3. Function: swapTokens**
  - Validate correct token swapping logic.
  - Ensure proper handling of slippage and price impact.
  - Check for reentrancy vulnerabilities.
  - Verify the correct calculation of swap amounts and fees.

5.5 MemeswapPairBase.sol

- **1. Function: initialize**
  - Validate correct initialization of pair base contracts.
  - Ensure proper handling of repeated initialization attempts.
  - Check for access control vulnerabilities.

5.6 MemeswapRouter.sol

- **1. Function: swapExactTokensForTokens**
  - Validate correct execution of token swaps.
  - Ensure proper handling of slippage and price impact.
  - Check for reentrancy vulnerabilities.
  - Verify the correct calculation of output amounts and fees.
- **2. Function: swapTokensForExactTokens**
  - Validate correct execution of token swaps for exact amounts.
  - Ensure proper handling of slippage and price impact.
  - Check for reentrancy vulnerabilities.
  - Verify the correct calculation of input amounts and fees.

5.7 MemeswapToken.sol

- **1. Function: createToken**
  - Validate correct creation of new tokens.
  - Ensure proper handling of token metadata and initialization.
  - Check for reentrancy vulnerabilities.
  - Verify that only authorized users can create tokens.

5.8 MemeswapVault.sol

- **1. Function: stake**
  - Validate correct staking of tokens.
  - Ensure proper calculation of staking rewards.
  - Check for reentrancy vulnerabilities.
  - Verify the correct emission of events during staking.

- **2. Function: unstake**
  - Validate correct unstaking of tokens.
  - Ensure proper handling of early unstaking penalties.
  - Check for reentrancy vulnerabilities.
  - Verify the correct emission of events during unstaking.

IRONNODE

ironnode.io

Thank you for choosing us!