IRONNODE

ironnode.io

SuperMeme

EVM Smart Contract Security Audit

by **IronNode** from **Oct 1 - Nov 1, 2024**

1	Executive Overview	3
1.1	Introduction	4
1.2	Audit summary	4
1.3	Findings and Recommendations	4
1.4	Audit Objectives	5
1.5	Scope	4
1.6	Assessment Summary & Findings Overview	6
2	Findings	7
2.1	Inconsistent Handling of Ether (msg.value) and _buyEth Parameter	8
	Code location	8
	Recommendation	9
2.2	Hardcoded Uniswap Router Address	10
	Code location	10
	Recommendation	10
2.3	Possible Denial of Service (DoS) Due to High Gas Consumption in refund Function	11
	Code location	11
	Recommendation	11
2.4	Incorrect Use of msg.sender Instead of _buyer in calculateUserBuyPointPercentages	12
	Code location	12
	Recommendation	12

2.5	Unfair Token Amount Adjustment Without Refund in buyTokens Function	13
	Code location	13
	Recommendation	13
2.6	Misuse of Slippage Parameter in buyTokens Function	14
	Recommendation	14
3	Manual testing	15-17

Executive overview

1 Executive Overview

1.1 Introduction

Supermeme is a decentralized platform designed to facilitate secure and customizable memecoin creation. Offering multiple token options — Refundable (rug-free), Locking Curve (vested), Dev-Locked, and Pure Degen — Supermeme allows creators to choose the level of risk and transparency suited to their projects.

The Refundable option ensures tokens can only be sold at the original purchase price during pre-liquidity, protecting early investors. The Locking Curve calculates token lock time based on purchase price, while Dev-Locked enables developers to lock their tokens for added security. Pure Degen offers a classic bonding curve for risk-tolerant traders.

With these structured choices, Supermeme minimizes financial risk, fostering trust among users. In response to a request from Supermeme, IronNode conducted a security audit of the platform from October 1 2024 to November 1 2024.

1.2 Audit summary

The security audit team at IronNode was allocated four weeks to conduct an extensive review of the Supermeme 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 Supermeme protocol, a total of six issues were identified, categorized as follows: one critical-severity issue, three medium-severity issues, and two low severity issue. Each of these issues was thoroughly documented and communicated to the Supermeme development team for remediation.

In response, the Supermeme 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.

The successful remediation of these issues underscores Supermeme 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.

1.4 Audit Objectives

- **1. Smart Contract Security**
  - **Code Review:** Identify vulnerabilities and enforce best practices in token creation options.
  - **Re-entrancy Protection:** Verify the presence of re-entrancy guards where applicable.
  - **Access Controls:** Ensure proper authorization mechanisms are in place for critical functions.
  - **Data Integrity:** Prevent vulnerabilities that could lead to data manipulation or loss.
  - **Arithmetic Errors:** Protect against integer overflow and underflow issues.

2. Token Creation Mechanisms

- **Refundable Option:** Ensure the rug-free bonding curve is secure and prevents financial loss for early investors.
- **Locking Curve:** Validate the locking mechanism to confirm accurate lock times based on purchase prices.
- **Dev-Locked Feature:** Assess the functionality and security of developer token locking for specified durations.
- **Pure Degen:** Ensure unrestricted trading during pre-liquidity is free from vulnerabilities.

3. Trader Security

- **Bonding Curves:** Ensure that bonding curves behave as expected and prevent unforeseen issues.
- **Transparency:** Validate that the platform provides clear information and terms for each token option.
- **Fair Trading Conditions:** Confirm that mechanisms in place protect traders from unexpected losses.

4. User Experience and Notifications

- **UI Signals and Notifications:** Enhance security by verifying that the UI provides accurate and clear signals to users.
- **User-Friendly Processes:** Ensure token creation and trading processes are easy to follow and secure.

5. Liquidity Security and Incentives

- **Liquidity Locking:** Confirm that liquidity locking mechanisms prevent manipulation and rug pulls.
- **Incentive Security:** Ensure reward and incentive mechanisms are fair, secure, and encourage platform trust.
- **Yield Distribution:** Validate accurate and secure distribution of rewards or incentives tied to liquidity and token holding.

1.5 Scope

Audit Scope:

supermeme-contracts
Branch: audit
Commit ID: 339cce6b7a74b0d19ac183369d181e7454c232e4

Verified Commit with Fixes:

supermeme-contracts
Branch: auditrev
Commit ID: 323a91243491b6da53def4d137a49c191375e2ab

1.6 Assessment Summary & Findings Overview

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	3	2	0

Security analysis	Risk Level	Remediation Status
Inconsistent Handling of Ether (msg.value) and _buyEth Parameter	CRITICAL	FIXED
Hardcoded Uniswap Router Address	MEDIUM	FIXED
Possible Denial of Service (DoS) Due to High Gas Consumption in refund Function	MEDIUM	FIXED
Incorrect Use of msg.sender Instead of _buyer in calculateUserBuyPointPercentages	MEDIUM	FIXED
Unfair Token Amount Adjustment Without Refund in buyTokens Function	LOW	FIXED
Misuse of Slippage Parameter in buyTokens Function	LOW	FIXED

Findings

2 Findings

2.1 Inconsistent Handling Of Ether (Msg.Value) And _buyEth Parameter

The buyTokens function in the contracts has inconsistent handling of msg.value (the Ether sent with the transaction) and the _buyEth parameter (the specified Ether amount to buy tokens). This inconsistency can lead to potential vulnerabilities, allowing attackers to manipulate the _buyEth and msg.value values, causing incorrect Ether calculations and creating opportunities for exploitation.

2.1.1 Code location

Function: buyTokens

```
require(minimumCost <= msg.value, "Insufficient funds");
require(_buyEth >= minimumCost && _buyEth <= totalCost + slippageAmount, "Slippage");
```

2.1.2 Recommendation

To mitigate this vulnerability, the _buyEth parameter should be removed, and all Ether calculations should be consistently based on msg.value. This change will ensure that there are no discrepancies between the Ether specified and the actual Ether sent.

Example Code Fix:

```
function buyTokens(uint256 _amount, uint256 _slippage) public payable nonReentrant {
    // Calculate cost and verify with msg.value consistently
    require(minimumCost <= msg.value, "Insufficient funds");
    // Remaining logic using msg.value only
}
```

This modification will ensure that all Ether-related calculations are consistent and prevent the manipulation of _buyEth and msg.value, enhancing the security of the contract's token purchase functionality.

2.1.3 Proof of Concept

An attacker can exploit this inconsistency by setting msg.value to cover the minimum cost required by buyTokens, while manipulating _buyEth to an inflated or deflated value, leading to unintended token purchasing behavior. This inconsistency allows the attacker to potentially complete the bonding curve without providing adequate funds.

```
    function testExploitBuyTokens() public {
        vm.startPrank(user1);
        uint256 user1Amount = 1000;
        uint256 user1Cost = degenBondingCurve.calculateCost(user1Amount);
        uint256 user1Tax = (user1Cost * tradeTax) / tradeTaxDivisor;
        uint256 user1TotalCost = user1Cost + user1Tax;
        uint256 user1Slippage = (user1TotalCost * 100) / 10000; // 1% slippage
        uint256 user1TotalCostWithSlippage = user1TotalCost + user1Slippage;

        degenBondingCurve.buyTokens{value: user1TotalCostWithSlippage}(
            user1Amount,
            100, // 1% slippage tolerance
            user1TotalCost
        );
        vm.stopPrank();

        // Now, the attacker executes the exploit
        vm.startPrank(attacker);

        // Amount of tokens to buy
        uint256 amount = 1000;

        // Calculate the legitimate cost of tokens
        uint256 cost = degenBondingCurve.calculateCost(amount);
        uint256 tax = (cost * tradeTax) / tradeTaxDivisor;
        uint256 totalCost = cost + tax;

        // Set slippage to maximum allowed (e.g., 10000 for 100%)
        uint256 slippage = 10000;

        // Calculate acceptable slippage range
        uint256 slippageAmount = (totalCost * slippage) / 10000;
        uint256 maxAcceptableEth = totalCost + slippageAmount;

        // Attacker sends only the totalCost as msg.value but claims to send inflatedBuyEth
        uint256 attackerMsgValue = totalCost;

        // Record initial balances
        uint256 attackerInitialBalance = attacker.balance;
        uint256 contractInitialBalance = address(degenBondingCurve).balance;

        // Perform the exploit
        degenBondingCurve.buyTokens{value: attackerMsgValue}(
            amount,
            slippage,
            maxAcceptableEth
        );

        degenBondingCurve.sellTokens(amount,0);
        vm.stopPrank();
    }
```

2.2 Hardcoded Uniswap Router Address

The contracts includes a hardcoded address for the Uniswap router, which could lead to potential issues in flexibility and security. This practice limits the contract's adaptability, as any future change in the router address would require redeploying the contract. Additionally, hardcoded addresses can increase the risk of human error and inadvertently connecting to a malicious or incorrect address, especially in testing or staging environments.

2.2.1 Code location

Function: Constructor

```
uniswapV2Router = IUniswapV2Router02(0x6682375ebC1dF04676c0c5050934272368e6e883);
```

2.2.2 Recommendation

To enhance flexibility and security, it is recommended to set the Uniswap router address during contract deployment as a constructor parameter or allow it to be updated through a secure, owner-restricted function. This approach will make it easier to update the router address if needed and reduce the risk of interacting with an unintended address.

Example Code Fix:

```
constructor(
    string memory _name,
    string memory _symbol,
    bool _devLocked,
    uint256 _amount,
    address _devAdress,
    address _revenueCollector,
    uint256 _devLockDuration,
    uint256 _buyEth,
    address _uniswapV2Router // Pass the Uniswap router address as a parameter
) public payable ERC20(_name, _symbol) {
    uniswapV2Router = IUniswapV2Router02(_uniswapV2Router);
    // Remaining constructor logic
}
```

Alternatively, if flexibility is required after deployment, consider adding a function restricted to the owner.

2.3 Possible Denial Of Service (DoS) Due To High Gas Consumption In Refund Function

The refund function in the SuperMemeRefundableBondingCurve contract contains nested loops, which could lead to high gas consumption as the number of participants and transactions increases. This high gas usage may exceed the block gas limit, preventing users from successfully calling the refund function and resulting in a Denial of Service (DoS) vulnerability.

2.3.1 Code location

Function: refund

```
function refund() public nonReentrant {
    // Further code...
    for (uint256 i = 0; i < userBuyPoints.length; i++) {
        uint256 buyPoint = userBuyPoints[i];
        uint256 ethPaidByOtherUsersInBetween = cumulativeEthCollected[buyCount]
- cumulativeEthCollected[buyPoint];

        for (uint256 j = buyCount; j >= buyPoint; j--) {
            if (buyIndex[j] == address(0)) {
                break;
            } else if (j == buyPoint) {
                break;
            } else if (
                buyIndex[j] == msg.sender || userRefunded[buyIndex[j]]
            ) {
                continue;
            } else {
                // Calculate and transfer refund amount for each eligible user
                uint256 refundAmountForInstance = (userBuyPointPercentages[buyIndex[j]][i]
* toBeDistributed) / buyPointScale;
                uint256 refundAmountForUser = (buyCost[j] * refundAmountForInstance)
/ ethPaidByOtherUsersInBetween;
                _transfer(address(this), buyIndex[j], refundAmountForUser);
            }
        }
    }

    userRefunded[msg.sender] = true;
    // Further code...
}
```

2.3.2 Recommendation

To address this issue, consider the following changes:

1. Optimize the Nested Loops: Refactor the logic to avoid iterating over every buy point and participant. For example, pre-calculate shared values outside of loops or use data structures that allow for more efficient lookups.
2. Enforce Minimum Buy Amount: Introduce a minimum buy amount to limit the number of participants, thus reducing the complexity and iteration count in the refund function.

2.4 Incorrect Use Of Msg.Sender Instead Of _buyer In CalculateUserBuyPointPercentages

The calculateUserBuyPointPercentages function mistakenly uses msg.sender instead of the _buyer parameter. This could lead to inconsistencies when msg.sender and _buyer differ, resulting in incorrect calculations of purchase point percentages. This issue may affect refund calculations, potentially leading to inaccurate refund amounts for users.

2.4.1 Code location

Function: calculateUserBuyPointPercentages

```
function calculateUserBuyPointPercentages(address _buyer) internal {
    uint256[] memory userBuyPoints = userBuysPoints[msg.sender];
    // Correct usage should be: userBuysPoints[_buyer];
    // ...
}
```

2.4.2 Recommendation

To ensure consistent behavior, replace msg.sender with the _buyer parameter within the function. This change will ensure that the function uses the correct context, as specified by the input parameter, making calculations accurate and consistent.

Example Code Fix:

```
function calculateUserBuyPointPercentages(address _buyer) internal {
    uint256[] memory userBuyPoints = userBuysPoints[_buyer];
    // Further code...
}
```

This modification will help prevent calculation errors by ensuring that the function operates consistently with the intended user context.

2.5 Unfair Token Amount Adjustment Without Refund In BuyTokens Function

In the buyTokens function, if the _amount specified by the user, when added to scaledSupply, exceeds MAX_SALE_SUPPLY, the contract adjusts _amount to fit within MAX_SALE_SUPPLY. However, it does not refund the excess Ether paid by the user. As a result, the user pays for a higher token amount than they actually receive, leading to an unfair trade and potential loss of funds for the user.

2.5.1 Code location

Function: buyTokens

```
if (scaledSupply >= MAX_SALE_SUPPLY) {
    bondingCurveCompleted = true;
    _amount = MAX_SALE_SUPPLY - (scaledSupply - _amount);
}
```

2.5.2 Recommendation

To ensure fairness, the contract should recalculate the costs based on the adjusted _amount and refund any excess Ether to the user. This can be achieved by recalculating cost, tax, and totalCost for the adjusted _amount and transferring the difference back to the user.

Example Code Fix:

```
if (scaledSupply + _amount > MAX_SALE_SUPPLY) {
    uint256 availableAmount = MAX_SALE_SUPPLY - scaledSupply;
    uint256 adjustedCost = calculateCost(availableAmount);
    uint256 adjustedTax = (adjustedCost * tradeTax) / tradeTaxDivisor;
    uint256 adjustedTotalCost = adjustedCost + adjustedTax;
    uint256 refundAmount = totalCost - adjustedTotalCost;

    cost = adjustedCost;
    tax = adjustedTax;
    totalCost = adjustedTotalCost;
    _amount = availableAmount;

    if (refundAmount > 0) {
        payable(msg.sender).transfer(refundAmount);
    }

    bondingCurveCompleted = true;
}
```

2.6 Misuse Of Slippage Parameter In BuyTokens Function

The _slippage parameter in the buyTokens function is currently ineffective and does not provide slippage protection as intended. Although _slippage is calculated in the function, it does not serve to limit price fluctuation tolerance for users, leaving them vulnerable to unexpected price changes. Instead, it incorrectly raises the minimum Ether required from the user without considering actual slippage.

2.6.1 Code location

Function: buyTokens

```
uint256 slippageAmount = (totalCost * _slippage) / 10000;
require(cost + tax <= msg.value, "Insufficient funds");
require(
    msg.value >= totalCost + slippageAmount,
    "Slippage"
);
```

2.6.2 Recommendation

To correctly implement slippage protection, modify the logic so that the _slippage parameter defines a tolerance range, setting an acceptable maximum Ether amount the user is willing to pay. This adjustment ensures that users are not charged more than their specified slippage tolerance allows.

Example Code Fix:

```
uint256 slippageAmount = (totalCost * _slippage) / 10000;
uint256 minimumCost = totalCost - slippageAmount;
uint256 maximumCost = totalCost + slippageAmount;

require(msg.value >= minimumCost && msg.value <= maximumCost, "Slippage exceeded");
```

In this corrected logic:

- minimumCost and maximumCost are calculated based on the user-defined _slippage tolerance.
- The require statement ensures that the Ether amount falls within the slippage tolerance, effectively protecting users against price fluctuations.

Manual testing

3 Manual testing

- **3.1 buyTokens Function**
  - Validate that only non-completed bonding curves allow token purchases.
  - Verify that _amount is greater than zero.
  - Ensure that _slippage parameter functions correctly, limiting price volatility as expected.
  - Test that the correct cost, tax, and totalCost values are calculated for the specified _amount.
  - Verify that if _amount + scaledSupply exceeds MAX_SALE_SUPPLY, _amount is adjusted, and excess Ether is refunded.
  - Check if totalEtherCollected, scaledSupply, and totalSupply values are updated correctly after a successful purchase.
  - Ensure that events tokensBought and Price are emitted with the correct data.
  - Test for potential reentrancy attacks, ensuring the nonReentrant modifier works as expected.

3.2 calculateCost Function

- Validate that the function accurately calculates the cost based on the given amount and current scaledSupply.
- Test various input values to ensure that calculations do not overflow and handle extreme values correctly.
- Confirm the cost calculation against expected bonding curve values.

3.3 payTax Function

- Verify that the _tax amount is transferred to the revenueCollector address correctly.
- Ensure that totalRevenueCollected is updated accurately after each tax payment.
- Check for integer overflow/underflow issues in _tax and totalRevenueCollected calculations.

3.4 sendToDex Function

- Ensure that bondingCurveCompleted is set to true before the function executes.
- Validate that sendDexRevenue is correctly transferred as tax to revenueCollector.
- Confirm that totalEtherCollected decreases by sendDexRevenue.
- Check that the liquidity added to the DEX includes the correct amount of ETH and tokens, matching totalEtherCollected.
- Verify that the SentToDex event is emitted with the correct ETH and token amounts.

3.5 sellTokens Function

- Validate that token selling is only allowed when bondingCurveCompleted is false.
- Check that _amount tokens are transferred back and burned from the seller’s balance.
- Verify that the correct refund, tax, and netRefund values are calculated.
- Ensure that totalEtherCollected, scaledSupply, and totalSupply are updated correctly after a sale.
- Confirm that the tokensSold and Price events are emitted with accurate data.
- Test for reentrancy vulnerabilities using the nonReentrant modifier.

- **3.6 calculateRefund Function**
  • Verify that the function correctly calculates the refund for a specified _amount of tokens.
  • Test various inputs to confirm that the refund calculation aligns with expected bonding curve logic.
  • Check for overflow and ensure the function handles large inputs without errors.

  3.7 _update Function (Token Transfer and Lock Mechanism)
  • Validate that _update allows transfers only when bondingCurveCompleted is true or within specified conditions.
  • Check that transfers from devAddress are restricted if devLocked is true and within devLockTime.
  • Confirm that super._update executes only when conditions allow.
  • Ensure rejections when transfer attempts do not meet conditions, and verify that appropriate events or errors are raised.
  • Test that only allowed transfers occur within the locked state, preventing unauthorized token transfers.

  3.8 Edge Cases and Other Tests
  • Test Completion: Confirm that bondingCurveCompleted is set to true when scaledSupply reaches MAX_SALE_SUPPLY.
  • Gas Optimization: Evaluate the gas cost for all functions, particularly for buyTokens and sellTokens, to ensure they stay within block gas limits.
  • Boundary Values: Test edge cases for token purchase and sale amounts close to zero and maximum supply to ensure proper functionality.
  • Error Handling: Check that each function throws appropriate errors and messages for invalid inputs or unauthorized actions.

IRONNODE

ironnode.io

Thank you for choosing us!