MEMED.FUN COMPREHENSIVE SECURITY AUDIT REPORT

Smart Contract Security Audit

Report prepared by:
Okiki Omisande
Blockchain Security Researcher
Twitter: @okiki_omisande
Telegram: @Invcbull
Date: September 10, 2025

Executive Summary

Project: Memed.fun - Meme Battle Platform

Audit Date: September 5-10, 2025

Auditor: INVCBULL AUDIT GROUP

Auditors Assigned: INVCBULL

Contracts Audited: MemedWarriorNFT.sol, MemedToken.sol, MemedFactory.sol, MemedEngageToEarn.sol, MemedBattle.sol

Total Findings: 10 vulnerabilities (1 Critical, 3 High, 2 Medium, 2 Low, 2 Informational)

Critical Risk Summary:

The Memed.fun platform contains a critical vulnerability that allows complete NFT theft, along with several high-severity issues that could lead to economic manipulation and system exploitation. **IMMEDIATE REMEDIATION REQUIRED** before any mainnet deployment.

Platform Overview: Memed.fun is a meme battle platform where users can create meme tokens, mint Warrior NFTs, and participate in battles to earn rewards. The platform integrates with Lens Protocol for social verification and uses a bonding curve mechanism for token launches.

CRITICAL SEVERITY FINDINGS

C-1: Complete NFT Theft Through Ownership Bypass

Location: `MemedWarriorNFT.sol:171-176` (allocateNFTsToBattle function)
Severity: CRITICAL
Impact: Complete theft of any NFT without ownership verification

Vulnerability Description:

The `allocateNFTsToBattle()` function in the MemedWarriorNFT contract lacks any ownership verification mechanism. When a user calls this function to allocate NFTs to a battle, the contract burns the specified NFTs and records the allocation without checking if the caller actually owns those NFTs. This flaw allows any user to target any NFT in the system that they don't own, burn it, and claim it as their own back if their supported meme wins the battle. The function only validates that the caller is the NFT contract itself (through the external modifier), but doesn't verify that the user parameter matches the actual owner of the NFTs being allocated.

Root Cause:

```
function allocateNFTsToBattle(uint256 _battleId, address _user, address _supportedMeme, uint256[] calldata _nftsIds) external {
    for (uint256 i = 0; i < _nftsIds.length; i++) {
        _burnWarrior(_nftsIds[i]); // □ No ownership check
    }
    memedBattle.allocateNFTsToBattle(_battleId, _user, _supportedMeme, _nftsIds);
}
```

Attack Path:

```
// Step 1: Alice owns NFT #123 (worth 5,000+ MEME tokens)
// Step 2: Bob calls allocateNFTsToBattle with Alice's NFT
allocateNFTsToBattle(battleId, bob, supportedMeme, [123]);

// Step 3: Alice's NFT #123 gets burned without her permission
// Step 4: NFT is allocated to Bob in the battle system
// Step 5: If Bob's meme wins, he calls getBackWarrior()
// Step 6: Bob receives Alice's NFT #123 as a new mint

// Result: Bob successfully steals Alice's NFT
```

Impact:

This vulnerability completely breaks the NFT ownership model by allowing any user to burn and steal any NFT without permission. The attack requires only gas fees and can target any NFT in the system, regardless of its value or the attacker's relationship to the owner. This undermines the fundamental trust required for users to participate in the battle system, as their valuable NFTs can be stolen at any time.

Fix:

```
function allocateNFTsToBattle(uint256 _battleId, address _user, address _supportedMeme, uint256[] calldata _nftsIds) external {
    for (uint256 i = 0; i < _nftsIds.length; i++) {
        require(ownerOf(_nftsIds[i]) == msg.sender, "Not the owner of NFT"); // ▯ Add ownership check
        _burnWarrior(_nftsIds[i]);
    }
    memedBattle.allocateNFTsToBattle(_battleId, _user, _supportedMeme, _nftsIds);
}

- this recommendation ensures that only the owner of the nft can allocate the nft to a battle
```

HIGH SEVERITY FINDINGS

H-1: Reentrancy in Engagement Rewards

Location: `MemedEngageToEarn.sol:64-68` (claimEngagementReward function)
Severity: HIGH
Impact: Double claiming of engagement rewards, reward pool drainage

Vulnerability Description:

The `claimEngagementReward()` function in the MemedEngageToEarn contract violates the checks-effects-interactions pattern by performing external token transfers before updating the state variables that track whether a user has already claimed their reward. This creates a classic reentrancy vulnerability where an attacker can call the function, receive the token transfer, and then call the function again before the state is updated to mark the reward as claimed. The function only checks the `isClaimedByUser` mapping at the beginning but doesn't update it until after the external call, creating a window for reentrancy attacks.

Root Cause:

```
function claimEngagementReward(uint256 _rewardId) external {
    require(!isClaimedByUser[_rewardId][msg.sender], "Already claimed");
    // ... calculate amount ...
    IERC20(reward.token).transfer(msg.sender, amount); // ▯ External call before state update
    isClaimedByUser[_rewardId][msg.sender] = true;     // ▯ State update after external call
    engagementRewards[_rewardId].amountClaimed += amount;
}
```

Attack Path:

```
// Step 1: Attacker calls claimEngagementReward
claimEngagementReward(rewardId);

// Step 2: External transfer triggers reentrancy
// Attacker's contract receives tokens and calls claimEngagementReward again

// Step 3: Second call succeeds because isClaimedByUser not yet updated
// Attacker claims same reward twice

// Result: Double claiming, reward pool drainage
```

Fix:

```
function claimEngagementReward(uint256 _rewardId) external {
    require(!isClaimedByUser[_rewardId][msg.sender], "Already claimed");
    // ... calculate amount ...

    // Update state first (checks-effects-interactions pattern)
    isClaimedByUser[_rewardId][msg.sender] = true;
    engagementRewards[_rewardId].amountClaimed += amount;

    // External call last
    IERC20(reward.token).transfer(msg.sender, amount);
}
```

H-2: Max Supply Bypass in Token Minting

Location: `MemedToken.sol:70-71` (claim function)
Severity: HIGH
Impact: Token supply can exceed 1B limit, breaking economic model

Vulnerability Description:

The `claim()` function lacks max supply enforcement, allowing the total token supply to exceed the intended 1 billion token limit, breaking the economic model and causing inflation.

Root Cause:

```
function claim(address to, uint256 amount) external {
    require(msg.sender == factoryContract, "Only factory can mint");
    // ❌ Missing: require(totalSupply() + amount <= MAX_SUPPLY, "Exceeds max supply");
    _mint(to, amount);
}
```

Attack Path:

```
// Step 1: Current supply approaches 1B limit
// Step 2: Factory calls claim() with large amount
claim(user, 100_000_000 * 1e18); // 100M tokens

// Step 3: Total supply exceeds 1B limit
// Economic model breaks, token value potentially diluted

// Result: Uncontrolled token inflation
```

Fix:

```
function claim(address to, uint256 amount) external {
    require(msg.sender == factoryContract, "Only factory can mint");
    require(totalSupply() + amount <= MAX_SUPPLY, "Exceeds max supply");
    _mint(to, amount);
}
```

H-3: Multiple LP Allocation Minting

Location: `MemedToken.sol:74-78` (mintUniswapLP function)
Severity: HIGH
Impact: LP allocation can be minted multiple times, breaking tokenomics

Vulnerability Description:

The `mintUniswapLP()` function lacks protection against multiple calls, allowing the 300M LP allocation to be minted multiple times.

Root Cause:

```
function mintUniswapLP(address to) external {
    require(msg.sender == factoryContract, "Only factory can mint");
    // ▯ No protection against multiple calls
    _mint(to, UNISWAP_LP_ALLOCATION); // 300M tokens each time
}
```

Attack Path:

```
// Step 1: Factory calls mintUniswapLP() first time
mintUniswapLP(factory); // Mints 300M tokens

// Step 2: Factory calls mintUniswapLP() again (bug or malicious)
mintUniswapLP(factory); // Mints another 300M tokens

// Step 3: Total LP allocation becomes 600M instead of 300M
// Tokenomics model breaks, excessive liquidity

// Result: Token supply inflation, economic model disruption
```

Fix:

```
bool private lpMinted = false;

function mintUniswapLP(address to) external {
    require(msg.sender == factoryContract, "Only factory can mint");
    require(!lpMinted, "LP already minted");

    lpMinted = true;
    _mint(to, UNISWAP_LP_ALLOCATION);
}
```

MEDIUM SEVERITY FINDINGS

M-1: Battle Reward Pool Inflation

Location: `MemedEngageToEarn.sol:74-78` (getBattleRewardPool function)
Severity: MEDIUM
Impact: Artificial inflation of battle reward pools

Vulnerability Description:

The `getBattleRewardPool()` function uses the contract's total token balance instead of tracking legitimate rewards, allowing attackers to artificially inflate battle rewards by transferring tokens to the contract.

Root Cause:

```
function getBattleRewardPool(address _token) external view returns (uint256) {
    uint256 balance = IERC20(_token).balanceOf(address(this)); // ⍰ Uses total balance
    return (balance * CYCLE_REWARD_PERCENTAGE) / 100; // 5% of total balance
}
```

Attack Path:

```
// Step 1: Attacker transfers tokens to contract
IERC20(memeToken).transfer(engageToEarnContract, 1000000 * 1e18);

// Step 2: getBattleRewardPool returns inflated amount
uint256 rewardPool = getBattleRewardPool(memeToken); // Now 5% of inflated balance

// Step 3: Battle rewards are artificially increased
// Legitimate users get more rewards than intended

// Result: Economic manipulation, unfair reward distribution
```

Fix:

```
mapping(address => uint256) public legitimateRewards;

function getBattleRewardPool(address _token) external view returns (uint256) {
    return (legitimateRewards[_token] * CYCLE_REWARD_PERCENTAGE) / 100;
}
```

M-2: Incorrect NFT Count for Engagement Rewards

Location: `MemedWarriorNFT.sol:277-287` (getWarriorMintedBeforeByUser function)
Severity: MEDIUM
Impact: Incorrect engagement reward calculations

Vulnerability Description:

The `getWarriorMintedBeforeByUser()` function only counts NFTs currently owned by the user, not historically minted NFTs. This allows users to mint NFTs, transfer them, and still receive engagement rewards for them.

Root Cause:

```
function getWarriorMintedBeforeByUser(address _user, uint256 _timestamp) external view returns (uint256) {
    uint256[] memory nfts = userNFTs[_user]; // ⍰ Only current ownership
    uint256 count = 0;
    for (uint256 i = 0; i < nfts.length; i++) {
        if (warriors[nfts[i]].mintedAt < _timestamp) {
            count++;
        }
    }
    return count; // Returns only currently owned NFTs
}
```

Attack Path:

```
// Step 1: User mints 10 NFTs
mintWarrior(); // 10 times

// Step 2: User transfers all NFTs to another address
transferFrom(user, otherAddress, tokenId); // For all 10 NFTs

// Step 3: User still gets engagement rewards for 10 NFTs
// Because function counts historically minted NFTs

// Result: User receives rewards without holding NFTs
```

Fix:

```
mapping(address => uint256) public totalMintedByUser;

function mintWarrior() external nonReentrant returns (uint256) {
    // ... existing code ...
    totalMintedByUser[msg.sender]++; // Track total minted
    // ... rest of function ...
}

function getWarriorMintedBeforeByUser(address _user, uint256 _timestamp) external view returns (uint256) {
    return totalMintedByUser[_user]; // Return total minted, not current ownership
}
```

LOW SEVERITY FINDINGS

L-1: Zero Address Validation Missing

Location: `MemedWarriorNFT.sol:63-67` (constructor)
Severity: LOW
Impact: Contract becomes unusable with zero addresses

Vulnerability Description:

The constructor lacks validation that immutable parameters are non-zero addresses, which could render the contract unusable if zero addresses are passed during deployment.

Root Cause:

```
constructor(
    address _memedToken,
    address _factory,
    address _memedBattle
) ERC721("Memed Warrior", "WARRIOR") {
    // □ No zero address validation
    memedToken = _memedToken;
    factory = IMemedFactory(_factory);
    memedBattle = MemedBattle(_memedBattle);
}
```

Fix:

```
constructor(
    address _memedToken,
    address _factory,
    address _memedBattle
) ERC721("Memed Warrior", "WARRIOR") {
    require(_memedToken != address(0), "Zero address not allowed");
    require(_factory != address(0), "Zero address not allowed");
    require(_memedBattle != address(0), "Zero address not allowed");

    memedToken = _memedToken;
    factory = IMemedFactory(_factory);
    memedBattle = MemedBattle(_memedBattle);
}
```

L-2: Missing totalClaimed Update

Location: `MemedEngageToEarn.sol:96-101` (claimBattleRewards function)
Severity: LOW
Impact: Inconsistent reward tracking

Vulnerability Description:

The `claimBattleRewards()` function doesn't update the `totalClaimed` mapping, leading to inconsistent tracking of claimed amounts across the system.

Root Cause:

```
function claimBattleRewards(address _token, address _winner, uint256 _amount) external {
    require(msg.sender == address(factory), "Only factory can transfer battle rewards");
    require(IERC20(_token).balanceOf(address(this)) >= _amount, "Insufficient balance");
    IERC20(_token).transfer(_winner, _amount);
    // □ Missing: totalClaimed[_token] += _amount;
}
```

Fix:

```
function claimBattleRewards(address _token, address _winner, uint256 _amount) external {
    require(msg.sender == address(factory), "Only factory can transfer battle rewards");
    require(IERC20(_token).balanceOf(address(this)) >= _amount, "Insufficient balance");

    totalClaimed[_token] += _amount; // □ Update tracking
    IERC20(_token).transfer(_winner, _amount);
}
```

INFORMATIONAL FINDINGS

I-1: Variable Naming Inconsistency

Location: `MemedWarriorNFT.sol:195-196` (getUserActiveNFTs function)
Impact: Code readability and maintainability

Issue:

```
function getUserActiveNFTs(address _user) external view returns (uint256[] memory) {
    uint256[] memory userTokens = userNFTs[_user]; // □ Inconsistent naming
    // Should be: uint256[] memory userNFTs = userNFTs[_user];
}
```

Fix:

```
function getUserActiveNFTs(address _user) external view returns (uint256[] memory) {
    uint256[] memory userNFTs = userNFTs[_user]; // ✅ Consistent naming
    // ... rest of function
}
```

I-2: Code Duplication

Location: `MemedWarriorNFT.sol:97-98` (mintWarrior function)
Impact: Code maintainability

Issue:

```
 // Duplicated fee calculation instead of using existing function
uint256 platformFee = (price * factory.platformFeePercentage()) / factory.feeDenominator();
// Should use: uint256 platformFee = calculateMintingFee(price);
```

Fix:

```
uint256 platformFee = calculateMintingFee(price); // ✅ Use existing function
```

RISK ASSESSMENT SUMMARY

Severity	Count	Risk Level
Critical	1	IMMEDIATE ACTION REQUIRED
High	3	HIGH PRIORITY
Medium	2	MEDIUM PRIORITY
Low	2	LOW PRIORITY
Informational	2	CODE QUALITY

FINANCIAL IMPACT ANALYSIS

Critical Findings:

- **C-1:** Complete NFT theft - Any NFT can be stolen without permission
- **Total at Risk:** All NFTs in the system (potentially millions in value)

High Findings:

- **H-1:** Reentrancy allows double claiming of engagement rewards
- **H-2:** Max supply bypass can cause unlimited token inflation
- **H-3:** LP allocation can be minted multiple times

Economic Impact:

- **NFT Theft:** Complete breakdown of NFT ownership model
- **Token Inflation:** Economic model disruption through supply manipulation
- **Reward Manipulation:** Unfair distribution of engagement and battle rewards

---

REMEDIATION SUGGESTED

Phase 1: Critical Fixes (IMMEDIATE)

1. **Fix C-1:** Add ownership verification to `allocateNFTsToBattle()`

2. **Fix H-1:** Implement checks-effects-interactions pattern in `claimEngagementReward()`
3. **Fix H-2:** Add max supply enforcement to `claim()` function
4. **Fix H-3:** Add protection against multiple LP minting

Phase 2: Medium Priority Fixes

6. **Fix M-1:** Track legitimate rewards separately from arbitrary transfers
7. **Fix M-2:** Track total minted NFTs instead of current ownership

Phase 3: Code Quality

8. **Fix L-1:** Add zero address validation to constructors
9. **Fix L-2:** Update totalClaimed tracking consistently
10. **Fix I-1:** Improve variable naming consistency
11. **Fix I-2:** Remove code duplication

TESTING RECOMMENDATIONS

Critical Path Testing:

- Test NFT ownership verification in battle allocation
- Verify reentrancy protection in reward claiming
- Test max supply enforcement with large mint amounts
- Test LP minting protection against multiple calls

Integration Testing:

- Full battle flow: NFT allocation → battle resolution → reward claiming
- Engagement reward claiming with various scenarios
- Token minting limits and supply tracking
- Economic model validation with edge cases

CONCLUSION

The Memed.fun platform contains **critical vulnerabilities** that must be addressed before any production deployment. The NFT theft vulnerability alone completely breaks the ownership model and could lead to massive economic losses. The additional high-severity issues around token supply control and reward manipulation further compound the risks.

Key Recommendations:

1. **DO NOT DEPLOY** until all Critical and High severity issues are resolved
2. **IMPLEMENT** comprehensive ownership verification across all NFT operations
3. **CONDUCT** thorough economic modeling to validate tokenomics
4. **ESTABLISH** proper supply controls and tracking mechanisms
5. **CONSIDER** formal verification for critical functions

The platform shows promise as a meme battle ecosystem, but requires significant security improvements before it can safely handle user funds and valuable NFTs.

Disclaimer