MyCut - Findings Report

Table of contents

Contest Summary

Sponsor: First Flight #23

Dates: Aug 29th, 2024 - Sep 5th, 2024

See more contest details here

Results Summary

Number of findings:

- High: 4
- Medium: 5
- Low: 8

High Risk Findings

H-01. Centralization risk for trusted owners

Description: The `ContestManager` contract has a centralization risk due to its reliance on the onlyOwner modifier. This means that the contract's critical functions can only be executed by the owner. This includes creating, funding, and closing contests, which centralizes control and decision-making power in a single entity or account.

Impact:

- *Single Point of Failure:* If the owner loses access to their account (e.g., private key compromise), the contract's functionality could be disrupted.
- *Malicious Actions:* The owner could potentially act maliciously, such as misappropriating funds or manipulating contest outcomes.
- *Lack of Transparency:* Participants in the contests may have reduced trust due to the centralized control.

Proof of Concept: The following functions in the ContestManager contract demonstrate centralization risk:

- `ContestManager::createContest`: Only the owner can create new contests.
- `ContestManager::fundContest`: Only the owner can fund contests.
- `ContestManager::closeContest`: Only the owner can close contests.

Example of centralization in code:

```
    function createContest(address[] memory players, uint256[] memory rewards,
IERC20 token, uint256 totalRewards)
        public
        onlyOwner
```

```
        returns (address)
    {
        // Owner-only action
    }

    function fundContest(uint256 index) public onlyOwner {
        // Owner-only action
    }

    function closeContest(address contest) public onlyOwner {
        // Owner-only action
    }
```

Recommended Mitigation: Decentralized Governance: Implement a multi-signature wallet or DAO (Decentralized Autonomous)

H-02. Unsafe ERC20 Operations should not be used in `ContractManager.sol`

Description: ERC20 functions may not behave as expected. For example: return values are not always meaningful.

Impact: If `ContractManager::transferFrom` fails and doesn't revert, the contract may behave incorrectly, assuming the transfer succeeded and it could lead to incorrect contest funding, loss of funds, or unexpected behavior.

Proof of Concept:

1. Use a token that doesn't revert or return false on failure.
2. Call fundContest with insufficient allowance or balance.
3. Observe that the contract proceeds without reverting, despite the transfer failing.

Recommended Mitigation: Follow the given below mitigation to avoid the problem:

1. Use OpenZeppelin's SafeERC20 library for safe token operations.
2. Replace `token.transferFrom(...)` with `token.safeTransferFrom(...)`.

H-03. Reentrancy concerns in the `ContestManager.sol`

Description: In the context of the `ContestManager.sol` contract, the primary concern would arise if the `Pot.sol` contract (which is interacted with by ContestManager) handles Ether or interacts with other external contracts in a way that could allow reentrant calls.

Impact:

1. *Unauthorized Access:* An attacker could exploit reentrancy to repeatedly call functions, potentially bypassing access controls or logic checks.
2. *State Manipulation:* The contract's state could be manipulated in unexpected ways, leading to incorrect contest results or fund distribution.
3. *Financial Loss:* If funds are involved, reentrancy could lead to unauthorized withdrawals or transfers.

Proof of Concept: Assuming the `Pot` contract has a vulnerability, here's a simplified example of how reentrancy might be exploited:

```
// SPDX-License-Identifier: MIT
    pragma solidity ^0.8.20;

    import "./ContestManager.sol";

    contract MaliciousContract {
        ContestManager contestManager;
        uint256 contestIndex;

        constructor(address _contestManager, uint256 _contestIndex) {
            contestManager = ContestManager(_contestManager);
            contestIndex = _contestIndex;
        }

        // Fallback function to exploit reentrancy
        fallback() external payable {
            if (address(contestManager).balance >= 1 ether) {
                contestManager.fundContest(contestIndex);
            }
        }

        function attack() external {
            contestManager.fundContest(contestIndex);
        }
    }
```

Recommended Mitigation:

1. *Implement Reentrancy Guards:*
    1. Use OpenZeppelin's ReentrancyGuard to protect functions that could be vulnerable to reentrancy attacks.
    2. Apply the nonReentrant modifier to functions in the Pot contract that handle token transfers or external calls.

```
    import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

    contract Pot is ReentrancyGuard {
        // Example function with reentrancy protection
        function distributeRewards() external nonReentrant {
            // Function logic
        }
    }
```

1. *Use SafeERC20 Library:*

    1. Replace direct ERC20 operations with OpenZeppelin's SafeERC20 library to handle token transfers safely.
    2. This ensures proper error handling and prevents silent failures.

```
    import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

    using SafeERC20 for IERC20;

    function fundContest(uint256 index) public onlyOwner {
        Pot pot = Pot(contests[index]);
        IERC20 token = pot.getToken();
        uint256 totalRewards = contestToTotalRewards[address(pot)];

        token.safeTransferFrom(msg.sender, address(pot), totalRewards);
    }
```

1. *Add Address Validation:*

    1. Ensure that addresses being assigned or used in critical operations are not address(0).
    2. Implement checks in functions to validate input addresses

H-04. Unchecked External Call in `Pot::_transferReward` function.

Description: The `Pot::_transferReward` function uses `i_token.transfer`, which is an external call that can fail if the token contract does not adhere to the ERC20 standard. This can lead to unexpected behavior if the transfer fails and is not handled properly.

Impact: If the token transfer fails, it could cause the contract to behave unexpectedly, potentially locking funds or causing incorrect state updates. This could result in users not receiving their rewards or the contract being unable to distribute remaining funds correctly.

Proof of Concept: The current implementation of the _transferReward function does not check the success of the transfer call:

```
    function _transferReward(address player, uint256 reward) internal {
        i_token.transfer(player, reward);
    }
```

If i_token is a non-standard ERC20 token that returns false instead of reverting, the transfer might silently fail, leaving remainingRewards inconsistent with actual token balances.

Recommended Mitigation: Use OpenZeppelin's SafeERC20 library, which provides a safeTransfer function that handles the return value correctly and reverts on failure. Modify the contract as follows:

```
    import {SafeERC20} from "lib/openzeppelin-
contracts/contracts/token/ERC20/utils/SafeERC20.sol";
    using SafeERC20 for IERC20;
```

```
    function _transferReward(address player, uint256 reward) internal {
        i_token.safeTransfer(player, reward);
    }
```

Medium Risk Findings

M-01. Missing checks for `address(0)` in the `ContestManager.sol` when assigning values to address state variables

Description: In Solidity, address(0) is often used to represent a null or uninitialized address. Failing to check for address(0) when assigning values to address state variables can lead to unintended behavior or vulnerabilities, as address(0) is typically not a valid or desired address in most contexts.

```
    contestToTotalRewards[address(pot)] = totalRewards;
```

Impact:

1. *Logic Errors:* Assigning address(0) could lead to incorrect logic execution, as it might represent an uninitialized or invalid state.
2. *Security Risks:* Functions that rely on valid address checks could be bypassed if address(0) is inadvertently used.
3. *Loss of Funds:* Sending Ether or tokens to address(0) results in irretrievable loss, as it acts as a burn address.

Proof of Concept: Consider a function that sets an owner address without checking for address(0):

```
    contract Example {
        address public owner;

        function setOwner(address newOwner) public {
            owner = newOwner; // No check for address(0)
        }
    }
```

In this example, calling setOwner(address(0)) would set the owner to an invalid address, potentially disrupting contract functionality.

Recommended Mitigation:

1. *Add Checks:* Implement checks to ensure that addresses being assigned are not address(0).

```
    function setOwner(address newOwner) public {
        require(newOwner != address(0), "Invalid address");
```

```
        owner = newOwner;
    }
```

1. Use Modifiers

M-02. `PUSH0` is not supported by all chains in the `ContestManager.sol`

Description: The PUSH0 opcode is a new Ethereum Virtual Machine (EVM) instruction introduced in the Shanghai upgrade. It pushes the constant value 0 onto the stack. However, not all Ethereum-compatible blockchains (e.g., some Layer 2 solutions or forks) have adopted this opcode, leading to compatibility issues.

Impact:

- *Compatibility Issues:* Contracts using PUSH0 may not deploy or execute correctly on chains that haven't implemented this opcode.
- *Deployment Failures:* Attempting to deploy contracts on incompatible chains could result in errors or failed transactions.
- *Reduced Portability:* Contracts are less portable across different EVM-compatible chains, limiting their usability and reach.

Proof of Concept: While PUSH0 is a low-level EVM instruction, its use can be indirectly observed in Solidity code compiled with newer compiler versions that target the Shanghai upgrade. Directly writing EVM bytecode is not typical in high-level Solidity contracts, but here's how you might encounter it:

```
    pragma solidity ^0.8.20;

    contract Example {
        function zero() public pure returns (uint256) {
            return 0; // This might compile to use PUSH0 in newer compilers
        }
    }
```

Recommended Mitigation:

- *Target Compatible Chains:* Ensure deployment targets are chains that support the Shanghai upgrade and PUSH0.
- *Use Compatible Compiler Versions:* Compile contracts with versions that do not introduce PUSH0

M-03. Incorrect Manager Cut Calculation in `Pot.sol`.

Description: The manager's cut is calculated using integer division: `remainingRewards / managerCutPercent`, which might not yield the intended percentage due to integer division truncation.

Impact: The manager might receive a smaller cut than intended, potentially leading to disputes or financial loss. The calculation could be misunderstood, leading to incorrect financial assumptions.

Proof of Concept: In the `Pot::closePot` function, the manager's cut is calculated as follows:

```
uint256 managerCut = remainingRewards / managerCutPercent;
```

If remainingRewards is 1000, the intended manager cut should be 100 (10%), but the logic might be misunderstood, and the calculation should be `remainingRewards * managerCutPercent / 100` for clarity and correctness.

Recommended Mitigation: Correct the calculation to ensure clarity and correctness:

```
uint256 managerCut = (remainingRewards * managerCutPercent) / 100;
```

M-04. Potential Integer Overflow in the `Pot::claimCut` function.

Description: The `Pot::claimCut` function updates the `remainingRewards` by subtracting the reward amount without explicitly checking for underflow. Although Solidity 0.8+ includes built-in overflow and underflow checks, ensuring logic correctness is crucial for clarity and future-proofing the code.

Impact: If there were any logic errors or changes in Solidity's behavior, subtracting a larger reward than `remainingRewards` could lead to an underflow, resulting in incorrect state updates and potentially allowing more rewards to be claimed than available.

Proof of Concept:

```
function claimCut() public {
    address player = msg.sender;
    uint256 reward = playersToRewards[player];
    if (reward <= 0) {
        revert Pot__RewardNotFound();
    }
    playersToRewards[player] = 0;
    remainingRewards -= reward; // Potential underflow if not checked
    claimants.push(player);
    _transferReward(player, reward);
}
```

If remainingRewards is less than reward, the subtraction would underflow in versions prior to Solidity 0.8. However, in Solidity 0.8+, this would revert automatically.

Recommended Mitigation: Although Solidity 0.8+ handles this automatically, it's a good practice to ensure the logic is clear and robust:

```
require(remainingRewards >= reward, "Insufficient remaining rewards");
remainingRewards -= reward;
```

M-05. Lack of access control on `Pot::claimCut`.

Description: The `Pot::claimCut` function allows any player to claim their reward, but there is no mechanism to prevent re-entrancy or multiple claims by the same player.

Impact: A player could potentially call claimCut multiple times if there were a re-entrancy vulnerability in the token contract, leading to multiple claims.

Recommended Mitigation: Although Solidity 0.8+ prevents re-entrancy by default, consider using a re-entrancy guard pattern or ensuring that the state is updated before any external calls:

```
    playersToRewards[player] = 0;
    remainingRewards -= reward;
    _transferReward(player, reward);
    claimants.push(player);
```

Low Risk Findings

L-01. Solidity pragma should be specific, not wide

Description: Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of pragma solidity ^0.8.0;, use pragma solidity 0.8.0; in file `ContestManager.sol` and `Pot.sol`

```
    pragma solidity ^0.8.20;
```

Impact: It will cost so much of much gas.

Recommended Mitigation: Try to provide the specific version of solidity in the contract so that il'll save the gas.

L-02. `public` functions not used internally could be marked `external`

Description: Instead of marking a function as public, consider marking it as external if it is not used internally in the `ContestManager.sol` and `Pot.sol`.

`ContestManager.sol`

```
    function createContest(address[] memory players, uint256[] memory rewards,
IERC20 token, uint256 totalRewards) public onlyOwner returns (address) {}
```

```
    function getContests() public view returns (address[] memory) {
        return contests;
    }
```

```
    function getContestTotalRewards(address contest) public view returns (uint256)
{
        return contestToTotalRewards[contest];
    }
```

```
    function getContestRemainingRewards(address contest) public view returns
(uint256) {
        Pot pot = Pot(contest);
        return pot.getRemainingRewards();
    }
```

`Pot.sol`

```
    function getToken() public view returns (IERC20) {
        return i_token;
    }
```

```
    function checkCut(address player) public view returns (uint256) {
        return playersToRewards[player];
    }
```

```
    function getRemainingRewards() public view returns (uint256) {
        return remainingRewards;
    }
```

L-03. Unused custom error in `Plot.sol`

Description: It is recommended that the definition be removed when custom error is unused.

```
    error Pot__InsufficientFunds();
```

L-04. Insufficient Storage Use in the `Pot.sol` contract.

Description: The contract uses separate arrays `i_players`, `i_rewards`, and `claimants`, which can lead to inefficient storage and gas usage.

Impact: While this does not pose a security risk, it can lead to increased gas costs for operations involving these arrays, especially as the number of players increases.

Recommended Mitigation: Consider using a struct to store player information, including their address and reward, in a single array. This can reduce storage complexity and improve gas efficiency.

L-05. Lack of Event Emissions in the `Pot.sol` contract.

Description: The contract does not emit events for critical operations such as claiming rewards and closing the pot.

Impact: Without events, it becomes difficult to track and audit contract activity off-chain, reducing transparency and making it harder to debug or verify actions.

Proof of Concept:

Recommended Mitigation: Emit events for key actions such as RewardClaimed and PotClosed to provide a clear audit trail.

```
    event RewardClaimed(address indexed player, uint256 reward);
    event PotClosed(uint256 remainingRewards, uint256 managerCut);

    function claimCut() public {
        ...
        emit RewardClaimed(player, reward);
    }

    function closePot() external onlyOwner {
        ...
        emit PotClosed(remainingRewards, managerCut);
    }
```

L-06. Hardcoded Manager Cut Percentage

Description: The manager cut percentage is hardcoded as a constant

```
    (managerCutPercent = 10).
```

Impact: This reduces flexibility, as any change to the manager's cut would require redeploying the contract, which can be costly and inconvenient.

Recommended Mitigation: Consider making the manager cut percentage a configurable parameter set during contract deployment or through a function callable by the owner. This allows for greater flexibility and adaptability to changing requirements.

L-07. Lack of input validations inside `Pot.sol` contract.

Description: The `constructor` does not validate the lengths of the players and rewards arrays to ensure they match.

Impact: If the arrays have mismatched lengths, it could lead to incorrect mappings in `playersToRewards`, potentially causing logical errors in reward distribution.

Recommended Mitigation: Add a check in the constructor to ensure that the lengths of players and rewards arrays are equal:

```
    require(players.length == rewards.length, "Players and rewards arrays must have the same length");
```

L-08. Potential Gas Limit Issues with Large Arrays

Description: Functions like `Pot::closePot` iterate over the `claimants` array, which could grow large.

Impact: If the array becomes too large, the transaction could exceed the block gas limit, making it impossible to execute.

Recommended Mitigation: Consider implementing a mechanism to process claims in batches or limit the number of claimants to ensure the function remains executable within gas limits.