PuppyRaffle Audit Report

Happy Man

Nov 20, 2024

Prepared by: Happy Lead Auditors: - Happy Man

Table of Contents

Protocol Summary

The **PuppyRaffle** smart contract presents a decentralized raffle system where participants pay an entry fee for a chance to win NFTs or other rewards. However, several critical security and efficiency issues have been identified, including:

1. **Reentrancy Attack Vulnerability**: The `refund` function is prone to reentrancy attacks because it doesn't follow the Checks-Effects-Interactions (CEI) pattern. This flaw allows malicious participants to drain the entire contract balance by exploiting reentrant calls

via a fallback function.
    - **Mitigation**: Update the `players` array before the external call to prevent reentrancy.
2. **Predictable Randomness in Winner Selection** : The current winner selection mechanism uses predictable inputs (`msg.sender`, `block.timestamp`, and `block.difficulty`), which can be manipulated by miners to influence the outcome.
    - **Mitigation**: Use Chainlink VRF (Verifiable Random Function) for secure and unpredictable randomness.
3. **Integer Overflow**: In older Solidity versions (pre-0.8.0), integer overflows can occur in variables like `totalFees`, which may result in incorrect fee calculations.
    - **Mitigation**: Upgrade to Solidity 0.8.0+ to utilize built-in overflow protection, or use SafeMath libraries in older versions.
4. **Gas Inefficiency Due to O(n²) Duplicate Checks** : The contract's `enterRaffle` function uses nested loops to check for duplicate participants, which increases gas costs and makes the system vulnerable to Denial-of-Service (DoS) attacks as the player base grows.
    - **Mitigation**: Replace the duplicate check with a `mapping(address => bool)` for constant-time efficiency.

Additional issues include outdated Solidity versions, unchecked state changes, and non-optimal gas usage, all of which need to be addressed for a secure and efficient contract.

Disclaimer

The Happy Man and our team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

```
./src/
└── PuppyRaffle.sol
```

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

I realy love to audit this kind of code base. I realy my best to do the work with it.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	1
Info	7
Gas	2
Total	13

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain all the reffle balance

Description: `PuppyRaffle::refund` don't follow the `CEI` rule

```
function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(
            playerAddress == msg.sender,
            "PuppyRaffle: Only the player can refund"
        );
        require(
            playerAddress != address(0),
            "PuppyRaffle: Player already refunded, or is not active"
        );

@>          payable(msg.sender).sendValue(entranceFee);
@>          players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

A player who entered the `PuppyRaffle::refund` raffle could have the `fallback/recevie` function and able to get the balance of the raffle till the balance reaches to zero.

Impact: All fees paid by participants could be stolen by malicious participants.

Proof of Concept:

User entered the `PuppyRaffle` and the attacker comes in the the contract having the fallback and receive fuction in there contract and the call the attack function and call the the `PuppyRaffle::refund` function and drain all the money from that contract.

Proof of Code:

▶ Code

Recommended Mitigation: To prevent this we should update the `PuppyRaffle::refunds` by updating the `players` array before calling the extarnal call. We should follow the `CEI` rule over there.

```
  function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(
            playerAddress == msg.sender,
            "PuppyRaffle: Only the player can refund"
        );
        require(
            playerAddress != address(0),
            "PuppyRaffle: Player already refunded, or is not active"
        );
+     players[playerIndex] = address(0);
+      emit RaffleRefunded(playerAddress);

        payable(msg.sender).sendValue(entranceFee);

-       players[playerIndex] = address(0);
-     emit RaffleRefunded(playerAddress);

    }
```

[H-2] Week randomness in `PuppyRaffle::selectWinner` allow users and minner to predict the winner.

Description: Hashing the `msg.sender` `block.timestamp` and `block.difficulty` make the predictable winner and controlled by the minner to choose the winner.

Impact: Minners and influence the winner for the raffle. and also select the `rarest` puppy.

Proof of Concept:

Validator are ahead to time by meanipulating the `block.timestamp` and `block.difficulty` and use to predict the when/how to participate see the solidity blog on prevrandso. `block.difficulty` is recently update with `prevrandao`.

Recommended Mitigation: Use the chainlink VRF to genrate the randomness.

[H-3] Integer overflow of `PupplyRaffle::totalFee` loos fee

Description: In solidity version prior to `0.8.0` the integer are subject to overflow.

```
uint64 myvar = type(64).max;
// => 18446744073709551615

myvar = myvar + 1 ;
// => 0
```

Impact: In the `PupplyRaffle::selectWinner` `totalFee` is accumulated in `feeaddress` to collect later in the withdraw fee however if the `totalFee` variable is overflow then the `feeaddress` may not able to collect the correct amount of fee. leaving fee parmanently stuck in contract.

Proof of Concept:

▶ Code

Recommended Mitigation: 1. Use the newer version of solidity and use the `uint256` instead of `uint64` in order to collect fee 2. If you want to use the older version of solidity then you need to use the `safemath` function form openzeplllin library. 3. Remove the balance check from the `puppyRaffle::withdrawFees`

```
-  require(    address(this).balance == uint256(totalFees), "PuppyRaffle:
```

this could be harmed by the `selfdestruct` function and then this require statament is gone wrong.

Medium

[M-1] Looping trough the `players` array for the duplicates checks in the `PuppyRaffle::enterRaffle` is a potential Denial of service (DoS) , incrementing gas cost for the future enterences.

Description: The `PuppyRaffle::enterRaffle` function currently checks for duplicate entries by iterating over the `players[]` array, which leads to an $O(n^2)$ time complexity as the number of participants grows. This method increases gas costs with each new entry and makes the contract vulnerable to Denial of Service (DoS) attacks by making transactions increasingly expensive over time. To optimize this, using a `mapping(address => bool)` for duplicate checks would provide constant time (O(1)) efficiency. This approach stores each player's address as a key and checks for duplicates in constant time, preventing gas cost escalation, enhancing scalability, and improving overall contract usability.

```
   // Check for duplicates
        for (uint256 i = 0; i < players.length - 1; i++) {
            for (uint256 j = i + 1; j < players.length; j++) {
                require(
                    players[i] != players[j],
                    "PuppyRaffle: Duplicate player"
                );
            }
        }
```

Impact: The `PuppyRaffle::enterRaffle` itrates through the `players[]` array which make more gas cost for the later users. The users that are comming into raffle first get the advantage over the other players how are entering latter in the raffle.

One more thing that the attacker can make the `PuppyRaffle::enterRaffle` array so big that the gas cost sky rokets and no any new users are able to get in the raffes and attacker try to make themselves winner.

Proof of concept:

if we have a two sets of entered players into raffle. Then the gas costs will be as such:

- 1st 100 players: ~6252128
- 2nd 100 players: ~18068218

3 times more expensive for the second 100 players.

▶ PoC

Recommended Mitigation:
To optimize your `enterRaffle` function by eliminating the duplicate checks with a mapping, you'll want to:

1. **Remove the nested loop for duplicate checks.**
2. **Introduce a mapping to track active players.**
3. **Add logic to update the mapping and emit the event only for new entries.**

Here's a cleaned-up version of your `enterRaffle` function with the suggested changes:

```
function enterRaffle(address[] memory newPlayers) public payable {
    require(
        msg.value == entranceFee * newPlayers.length,
        "PuppyRaffle: Must send enough to enter raffle"
    );

    for (uint256 i = 0; i < newPlayers.length; i++) {
        require(!activePlayers[newPlayers[i]], "PuppyRaffle: Duplicate
player");
        players.push(newPlayers[i]);
        activePlayers[newPlayers[i]] = true; // Add to mapping
    }

    emit RaffleEnter(newPlayers);
}
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` return the zero index for non-existing players but if the user entered first and they have the zeroth index and user may think that they might we in active players.

Description: If the player in `PuppyRaffle::getActivePlayerIndex` is at index 0 , then it will return the zero index , but according to netspec if there is no active player in `PuppyRaffle::getActivePlayerIndex` it will also return the zero index.

Impact: A player at Index 0 is incorrectly think that he is still not entered the raffle and try to do this again , which is nothing but the waste of gass fee.

Recommended Mitigation: If the player in the `PuppyRaffle::getActivePlayerIndex` is not exist then the better option might be return the revert or also use the int256 to return the -1 instead.

Gas

[G-1] Unchanged state variables should be declared as constants or immutable

Discription Reading from storage is much more expensive than reading from immutable storage and constants.

- Instances:
- `puppyRaffle::raffleDuration` should be `immutable`
- `puppyRaffle::commonImgUri` should be `constand`
- `puppyRaffle::rareImageUri` should be `constant`
- `puppyRaffle::legendryImageUri` should be `constant`

[G-2] Storage variables in a loop should be cahced

Discription Eveytime we read from `player.length` in loop it reads from the storage which should be more gass costly then the memory so modify the lood as follows:

```
+   uin256 players = players.length
-  for (uint256 i = 0; i < players.length - 1; i++) {
-           for (uint256 j = i + 1; j < players.length; j++) {
-               require(
-                   players[i] != players[j],
-                   "PuppyRaffle: Duplicate player"
-               );
-           }
-       }
```

Informational

[I-1]: Solidity pragma should be specific, not wide

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;`

▶ 1 Found Instances

[I-2] Using a outdated version of solidity is not recommended

Description: solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation: Deploy with a recent version of Solidity (at least `0.8.0`) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

follow the slither documentation for more information

[I-3] Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

▶ 2 Found Instances

[I-4] `PuppyRaffle::selectWinner` does not follow the `CEI` which is not a best practice

[I-5] Use of magic numbers is discouraged

[I-6] State change missing events , every time when state changes there must be an event emitted

[I-7] `PuppyRaffle::_isActivePlayer` is never used , so it's safe to remove it