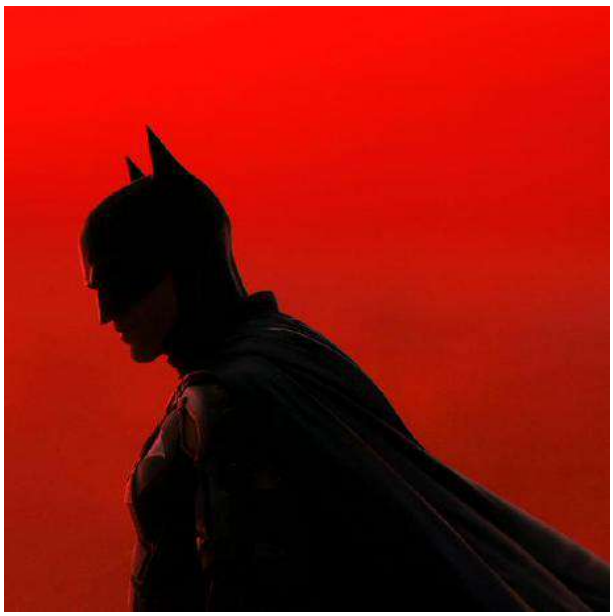

PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

August 18, 2025

Protocol Audit Report

Alqasem Hasan

August 18, 2025

Prepared by: Alqasem Hasan Lead Auditors:

- Alqasem Hasan

Table of Contents

- Medium
  - [M-1] In the `PuppyRaffle::enterRaffle` function, due to looping over the unbounded array players, there is a risk of denial of service attack.
  - [M-2] Typecasting from uint256 to uint64 in PuppyRaffle.selectWinner() May Lead to Overflow and Incorrect Fee Calculation
  - [M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest
- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing the player at index 0 to think they are not in the raffle.
- Gas
  - [G-1] Unchanged state variables should be declared constant or immutable
  - [G-2] Storage variables in a loop should be cached
- Informational/Non-Crits
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2] Using an outdates version of Solidity is risky
  - [I-3] Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` should follow CEI, which is not a best practice
  - [I-5] Use of "magic" numbers is discouraged
  - [I-6] State changes are missing events
  - [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed

- Additional Findings not taught in the course yet
  - MEV

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 Alqasem 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
Likelihood	High	H	H/M	M
	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 loved auditing this codebase. Patrick is such a sigma boy at writing intentionally bad code.

Issues found

Issues found

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

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` function allows someone to drain the contract of funds.

Description: The `PuppyRaffle::refund` function does not follow CEI (Checks-Effects-Interactions) and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making the external call do 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 is drained of funds.

Impact: All fees paid by the raffle contract could be stolen by the maalicous participant.

Proof of Concept:

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

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
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);

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

        uint256 balanceBefore = address(reentrancyAttacker).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;
        // attack
```

```
16          vm.prank(attackUser);
17          reentrancyAttacker.attack{value: entranceFee}();
18
19          console.log("Starting attacker contract balance :",
                balanceBefore);
20          console.log("Starting contract balance :",
                startingContractBalance);
21
22          uint256 balanceAfter = address(reentrancyAttacker).balance;
23          console.log("Balance of reentrancy attack after:", balanceAfter
                );
24          console.log("Balance of puppy raffle after:", address(
                puppyRaffle).balance);
25      }
```

and this contract as well:

```
1  contract ReentrancyAttack {
2      PuppyRaffle public puppyRaffle;
3      uint256 entranceFee;
4      uint256 attackerIndex;
5
6      constructor(PuppyRaffle _puppyRaffle) {
7          puppyRaffle = _puppyRaffle;
8          entranceFee = puppyRaffle.entranceFee();
9          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
10     }
11
12     function attack() public payable {
13         address[] memory players = new address[](1);
14         players[0] = address(this);
15         puppyRaffle.enterRaffle{value: entranceFee}(players);
16         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
17         puppyRaffle.refund(attackerIndex);
18     }
19
20     receive() external payable {
21         if (address(puppyRaffle).balance >= entranceFee) {
22             puppyRaffle.refund(attackerIndex);
23         }
24     }
25 }
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle:refund` function update the `players` array before making the external call. Additionally, 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 the winner and influence or predict the winner puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable random number. A predictable random numnber is not a random number, and can be manipulated by miners to get the best NFT!

Note: This means users could front-run this function and call `refund` 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 useless 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. See the solidity blog on prevrando was recently replaced with 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 their `selectWinner` transaction if they don't like the winner or resulting puppy.

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

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

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses 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;
// myVar will be 0
```

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

Proof of Concept:

1. We simulat the raffle for 30 rounds, with 100 players each time.
2. We calculate the expected total fees, which is 600 ETH.
3. We then check the actual total fees.

```
totalFees = totalFees + uint64(fees);
// aka
totalFees = 80000000000000000 + 178000000000000000;
// and this will overflow!
totalFees = 153255926290448384;
```

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match andwithdraw 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 the above `require` will be impossible to hit.

Code

```

function test_ArithmeticOverflow() public {
        uint256 playersNum = 100;
        uint256 totalAmount = 0;
        uint256 expectedTotalFee = 0;

        // Simulate the raffle with mutliple entries from players
           arrays and with simulated time passing
        for (uint256 i = 0; i < 30; i++) {
            address[] memory players = new address[](playersNum);
            for (uint256 j = 0; j < playersNum; j++) {
                players[j] = address(j + i * 100); // Create unique
                   addresses
            }
            // Manually calculate the actual total fee
            totalAmount += entranceFee * players.length;
            expectedTotalFee = (totalAmount * 20) / 100;

            // Enter the raffle with players
```

```
18              puppyRaffle.enterRaffle{value: entranceFee * players.length
                    }(players);
19
20              // Simulate time passing to select a winner
21              vm.warp(block.timestamp + duration + 1);
22              vm.roll(block.number + 1);
23
24              // select a winner for the raffle
25              puppyRaffle.selectWinner();
26          }
27
28          console.log("Expected total fees");
29          console.logUint(expectedTotalFee);
30          // 600.000000000000000000 ETH
31
32          uint256 actualTotalFees = puppyRaffle.totalFees();
33
34          console.log("Actual total fees");
35          console.logUint(actualTotalFees);
36          // 9.704189641294348288 ETH
37
38          assert(expectedTotalFee > actualTotalFees);
39
40          // 0.800000000000000000 ETH
41
42          // This will cause an overflow if totalFees is uint64
43          // The max uint64 is 18446744073709551615
44          // This is 18.446744073709551615 eth
45          // This is the number you get when you type cast 20 eth to a
                uint64.
46          // you get 1.553255926290448384 ETH
47      }
```

Recommended Mitigation: There are a few possible mitigations:

1. Use a newer version of Solidity, and a `uint256` instead of a `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of openzeppelin for verison 0.7.6 of solidity, however you would still have a hard time with the `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 attack vectors with that final require, so we recommend removing it regardless.

Medium

[M-1] In the `PuppyRaffle::enterRaffle` function, due to looping over the unbounded array players, there is a risk of denial of service attack.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, as more players are added, the gas cost of checking for duplicates increases, which makes it more expensive for users call this function, the later they join the raffle. Every additional address added to the `players` array, means an additional check he loop will have to make, and a greater gas cost. This could lead to a denial of service attack, where an attacker could add many players to the raffle, making the `PuppyRaffle::enterRaffle` function too expensive to call for new players.

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

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users 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::entrants` array so big, that no one else enter, guaranteeing that they will win the raffle, as they are the only player in the raffle.e

Proof of Concept:

If we have 2 sets of 100 players to enter, the gas costs will be as such:

- 1st 100 players: ~ 6503275
- 2nd 100 players: ~ 18995515

This is more than 3x more expensive to enter the raffle for the second group of players.

PoC

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

```
function testDenialOfServiceEnterRaffle() public {
        vm.txGasPrice(1);
        // Create the first list of players to enter the raffle
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
```

```
7               players[i] = address(i); // Create unique addresses
8           }
9           // Record the gas used for the first entry
10          uint256 gasStart = gasleft();
11          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
12          uint256 gasEnd = gasleft();
13          uint256 gasUsedFirst = gasStart - gasEnd;
14          console.log("Gas used for first entry:", gasUsedFirst);
15
16          // Create the second list of players to enter the raffle
17          address[] memory playersTwo = new address[](playersNum);
18          for (uint256 i = 0; i < playersNum; i++) {
19              playersTwo[i] = address(i + playersNum); // Create unique
                    addresses
20          }
21          // Record the gas used for the second entry
22          uint256 gasStartSecond = gasleft();
23          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                playersTwo);
24          uint256 gasEndSecond = gasleft();
25          uint256 gasUsedSecond = gasStartSecond - gasEndSecond;
26          console.log("Gas used for first entry:", gasUsedSecond);
27
28          // Assert that the gas used for the second entry is more than
                the first, and if this is true, then that means it is more
                expensive for people to enter the raffle the later they
                enter, hence DoS.
29          assert(gasUsedFirst < gasUsedSecond);
30      }
```

Recommended Mitigation: There are a few reccomendations.

1. Consider allowing duplicate players. Users can make new wallet addresses anyways, so a duplicate check does not prevent the same person from entering the raffle multiple times from different addresses, it only prevent the same wallet address.
2. Consider using a mapping to check for duplicates, this would allow constant time lookup of whether a user has already entered.

```
1  + mapping(address => uint256) public addressToRaffleId;
2  + uint256 public raffleId = 0;
3      .
4      .
5      .
6  function enterRaffle(address[] memory newPlayers) public payable {
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 -     // Check for duplicates
13 +       // Check for duplicates only from the new players
14 +       for (uint256 i = 0; i < newPlayers.length; i++) {
15 +             require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
16 +}
17 -             for (uint256 j = i + 1; j < newPlayers.length; j++) {
18 -                 require(newPlayers[i] != newPlayers[j], "PuppyRaffle:
       Duplicate player");
19 -         }
20 -       }
21 -         for (uint256 i = 0; i < players.length - 1; i++) {
22 -             for (uint256 j = i + 1; j < players.length; j++) {
23 -                 require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
24 -             }
25 -         }
26 -         emit RaffleEnter(newPlayers);
27 -     }
28
29 .
30 .
31 .
32      function selectWinner() external {
33 +       raffleId++;
34          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

3. Alternatively, you could use OpenZeppelin's `EnumerableSet` library, which allows you to store a set of unique values, and provides constant time lookup for whether a value is in the set.

[M-2] Typecasting from uint256 to uint64 in PuppyRaffle.selectWinner() May Lead to Overflow and Incorrect Fee Calculation

Description: The type conversion from uint256 to uint64 in the expression 'totalFees = totalFees + uint64(fee)' may potentially cause overflow problems if the 'fee' exceeds the maximum value that a uint64 can accommodate (2^64 - 1).

```
1 totalFees = totalFees + uint64(fee);
```

Impact: This could consequently lead to inaccuracies in the computation of 'totalFees'.

Proof of Concept:

Code

```
function testOverflow() public {
        uint256 initialBalance = address(puppyRaffle).balance;

        // This value is greater than the maximum value a uint64 can
            hold
        uint256 fee = 2**64;

        // Send ether to the contract
        (bool success, ) = address(puppyRaffle).call{value: fee}("");
        assertTrue(success);

        uint256 finalBalance = address(puppyRaffle).balance;

        // Check if the contract's balance increased by the expected
            amount
        assertEq(finalBalance, initialBalance + fee);
    }
```

In this test, assertTrue(success) checks if the ether was successfully sent to the contract, and assertEq(finalBalance, initialBalance + fee) checks if the contract's balance increased by the expected amount. If the balance didn't increase as expected, it could indicate an overflow.

Recommended Mitigation: To resolve this issue, you should change the data type of `totalFees` from `uint64` to `uint256`. This will prevent any potential overflow issues, as `uint256` can accommodate much larger numbers than `uint64`. Here's how you can do it:

Change the declaration of `totalFees` from:

```
uint64 public totalFees = 0;
```

to:

```
uint256 public totalFees = 0;
```

And update the line where `totalFees` is updated from:

```
- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;
```

This way, you ensure that the data types are consistent and can handle the range of values that your contract may encounter.

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

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

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

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

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

Proof of Concept: (or proof of code)

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

Recommended Mitigation:

1. Restrict access to the raffle to only EOAs (Externally Owned Accounts), by checking if the passed address in enterRaffle is a smart contract, if it is we revert the transaction.
2. Create a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize. (Recommended)

Pull > Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing the player at index 0 to think they are not in the raffle.

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 raffle.

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

```
4                 if (players[i] == player) {
5                     return i;
6                 }
7             }
8             return 0;
9         }
```

Impact: A player at index 0 may incorrectly think they are not in the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function 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 the 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 constant or immutable

Reading from storage is much more expensive that reading from 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 read from storage, as opposed to memory which is more gas efficient.

```
+       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++) {
```

```
5 +             for (uint256 j = i + 1; j < playerLength; j++) {
6 +               require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
7               }
8           }
```

Informational/Non-Crits

[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;`

- Found in src/PuppyRaffle.sol Line: 2

[I-2] Using an outdates version of Solidity is risky

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.

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.

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

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 204

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI, which is not a 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 in a codebase, and it's much more readable if the numbers are given a name.

Examples:

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

Instead, you could use:

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

[I-6] State changes are missing events

Events are a great way to track state changes in a contract. They are not required, but they are a best practice. Events in this contract are also not using indexed paramters, which would allow for easier filtering of events.

You can add an event for the `selectWinner` function that emits the winner and the puppy they won.

```
+    event RaffleWinner(address indexed winner, uint256 indexed tokenId, string puppyType);
```

```
-         emit RaffleEnter(newPlayers);
+         event RaffleEnter(address[] players, address indexed entrant);
```

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

This function is internal, which means it is not used outside of the contract, and it is never called in the contract. It should be removed to keep the code clean.

```
1 -    function _isActivePlayer(address player) internal view returns (
  bool) {
2 -        for (uint256 i = 0; i < players.length; i++) {
3 -            if (players[i] == player) {
4 -                return true;
5 -            }
6 -        }
7 -        return false;
8 -    }
```

Additional Findings not taught in the course yet

MEV