Puppy Raffle Audit Report

Version 1.0

Bricks

February 3, 2026

Protocol Audit Report

Bricks

February 3, 2026

Prepared by: Bricks Lead Auditors: - Bricks

Table of Contents

  - [M-1] TITLE Making the `puppyRaffle::totalFees` variable a `uint64` and always adding fees generated from players entrance fee to it can cause an overflow and revert the `puppyRaffle::selectWinner` function call.

- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` function returns `0` if the player is not active. This could be misleading, as it is also a valid index.

- Informational
  - [I-1] Solidity pragma should be specific not wide
  - [I-2] Using an outdated version of solidity is not recomended
  - [I-3] Check for address(0) when assigning values to address state variables.
  - [I-4] `PuppyRaffle::selectWinner` function does not follow the CHECKS-EFFECTS-INTERACTIONS pattern
  - [I-5] Use of magic numbers is not recommended as its meaning is unclear
  - [I-6] `PuppyRaffle::_isActivePlayer` internal function is never used in the contract.

- Gas
  - [G-1] Unchanged state variables should be declared as constant or immutable
  - [G-2] Storage variable in a loop should be cached in memory

Protocol Summary

This is a raffle protocol that allows users to enter and win a random puppy NFT. The users can call the enter raffle function to enter the raffle but cant enter duplicate addresses. Anyone can call the refund function to get their ticket back if they want to. The owner of the contract can set a fee address and the rest of the funds will be sent to the winner of the raffle.

Disclaimer

The Bricks 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

- In 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

For this audit a proper scoping of the contracts was done and the findings were limited to the ones that were found in the `puppyRaffle.sol` file and a the protocol test coverage was done inorder to see which part was priotized first. The findings were then sorted by severity and with the help of some static analysis tools new findings were found.

Issues found

severity	Number of issues found
High	3
Medium	1
Low	1
Informational	6
Gas	2
Total	13

Findings

High

[H-1] TITLE Making external calls in before state changes in the `puppyRaffle::refund` function leads to a reentrancy vulnerability.

Description: The `puppyRaffle.refund` function makes external calls to other contracts or address before state changes. This can lead to a reentrancy attack and an attacker can keep calling the `puppyRaffle.refund` function without any impact on his balance until the contract is drained of its funds.

```
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            // @audit Reentrancy vulnerability exists here due to the
                 external call before state changes.
6  @>          payable(msg.sender).sendValue(entranceFee);
7
8          players[playerIndex] = address(0);
9          emit RaffleRefunded(playerAddress);
10     }
```

Impact: An attacker can keep calling the `puppyRaffle::refund` function without any impact on his balance until the contract is drained of its funds.

Proof of Concept: This example shows how an attacker can keep calling the `puppyRaffle::refund` function without any impact on his balance until the contract is drained of its funds. In the end the puppyRaffle contract is left with zero balance. Here is a testcase that shows how an attacker can keep calling the `puppyRaffle::refund` function without any impact on his balance until the contract is drained of its funds.

PoC

```
1      function testRefundReentrancyAttack() public playerEntered {
2          // Deploy the attack contract
3          ReentrancyAttack attackContract = new ReentrancyAttack(address(
               puppyRaffle), entranceFee);
4
5          address[] memory players = new address[](10);
6          for (uint256 i = 0; i < players.length; i++) {
7              players[i] = address(uint160(i + 200));
8          }
9          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
               players);
10
11         // Fund the attack contract
12         vm.deal(address(attackContract), entranceFee);
13
14         // Execute the attack
15         address[] memory playersAttack = new address[](1);
16         playersAttack[0] = address(attackContract);
17         attackContract.attack{value: entranceFee * playersAttack.length
               }(playersAttack);
18         assertEq(address(puppyRaffle).balance, 0);
19     }
```

The test above passes and proofs that the `puppyRaffle::refund` function is vulnerable to reentrancy attacks.

Recommended Mitigation: To prevent reentrancy attacks in the `puppyRaffle::refund` function, a classic CHECKS-EFFECTS-INTERACTIONS pattern can be used where for every function checks should come first and then effects like state variables changes should come second. This ensures no interactions are happening between the checks and the effects. A second alternative is to use the openzeppelin's `ReentrancyGuard` contract which will prevent reentrancy attacks.

```
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  +        players[playerIndex] = address(0);
```

```
7
8           payable(msg.sender).sendValue(entranceFee);
9
10 -         players[playerIndex] = address(0);
11          emit RaffleRefunded(playerAddress);
12      }
```

[H-2] TITLE hashing on-chain data as a source of randomness is a weak way to generate random numbers. Leading to minners posibly predicting the winner of the raffle.

Description: Hashing on-chain data in as a source of randomness is a weak way to generate randomness. In the `puppyRaffle::selectWinner` function, on-chain data like `msg.sender`, `block.timestamp` and `block.prevrandao` are all hashed using the `keccak256` function. This can lead to minners posibly predicting the winner of the raffle.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
4          // @audit Weak source of randomness
5          uint256 winnerIndex =
6  @>              uint256(keccak256(abi.encodePacked(msg.sender, block.
       timestamp, block.prevrandao))) % players.length;
7          address winner = players[winnerIndex];
8          // q: why not just do address(this).balance?
9          uint256 totalAmountCollected = players.length * entranceFee;
```

Impact: Minners posibly predicting the winner of the raffle or making sure they pick the best scenario where they will win since they can see add the trabsaction at times where the resulting hash modulo of the players length gives their address.

Recommended Mitigation: A better way to generate randomness is to use Chainlink VRF for cryptographically secure and provably random values to ensure protocol integrity. Using chainlink VRF gauantees a way where random number are generated off-chain and delivered on-chain by the chainlink VRF nodes.

[H-3] TITLE The `puppyRaffle::withdrawFees` function checks contract balance equal total fees to ensure players are not in raffle. this can cause ethers to be mishandled.

Description: The `puppyRaffle::withdrawFees` function checks contract balance equal total fees to ensure players are not in raffle. Even if contract lacks recieve or fallback function which is going

to make sure another contract or user dont send it ether it is still possible for other contract to send the `puppyRaffle` ether using the `selfdestruct` function.

```
    function withdrawFees() external {
     // @audit mishandling of ether since this contract can be forced to recieve ether using the selfdestruct function
@>       require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
         uint256 feesToWithdraw = totalFees;
         totalFees = 0;
         (bool success,) = feeAddress.call{value: feesToWithdraw}("");
         require(success, "PuppyRaffle: Failed to withdraw fees");
     }
```

Impact: The `puppyRaffle::withdrawFees` function can be rendered useless since the contract can be forced to recieve ether using the `selfdestruct` function which causes the check `require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");` to revert. This can cause a loss of ether in fees to the contract owner.

Proof of Concept: The example below shows how the `puppyRaffle::withdrawFees` function can be rendered useless due to the mishandling of ether.

PoC

```
    function testMishandlingOfEther() public {
        BruteForceEtherSend forceEtherIn = new BruteForceEtherSend(address(puppyRaffle));
        uint256 numOfPlayers = 10;
        address[] memory players = new address[](numOfPlayers);
        for (uint256 i = 0; i < players.length; i++) {
            players[i] = address(uint160(i + 300));
        }
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();

        forceEtherIn.Attack{value: 1e18}();

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

The test above test passes even when players are not in raffle and proves that the `puppyRaffle::withdrawFees` function is vulnerable to mishandling of ether.

Recommended Mitigation: In order to avoid this vulnerability, a better way to check if players are in the raffle should be implemented. This can be done by checking if the `puppyRaffle::players` array is empty since the select winner always clear the `puppyRaffle::players` array.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
+       require(players.length == 0, "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

This way any forced in ether dosen't distrupt the protocol operation and doing so will be in benefit of the contract owner and users of the contract. Note: The selfdestruct function can be used to force ether to be sent to the contract but no longer delete the contract since the cancun hardfork.

Medium

[M-1] TITLE Making the `puppyRaffle::totalFees` variable a `uint64` and always adding fees generated from players entrance fee to it can cause an overflow and revert the `puppyRaffle::selectWinner` function call.

Description: The `puppyRaffle::selectWinner` function always calculate the fees as 20% of the cumulative entrance fee for all players and increament the `puppyRaffle::totalFees` by the fee value gotten from that raffle round. Since the total fees is of a `uint64` once fees reaches slightly above 18 ether an overflow occurs and every `puppyRaffle::selectWinner` call will revert until all fees are withdrawn.

```
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.prevrandao))) % players.length;
        address winner = players[winnerIndex];
        // q: why not just do address(this).balance?
        uint256 totalAmountCollected = players.length * entranceFee;
        // q: is there a reason for prizePool being 80% and fee being 20%?
```

```
10          uint256 prizePool = (totalAmountCollected * 80) / 100;
11          uint256 fee = (totalAmountCollected * 20) / 100;
12
13          // @audit an overflow can occur once the total fees fills the
                uint64 max storage value
14  @>        totalFees = totalFees + uint64(fee);
15  `
16          uint256 tokenId = totalSupply();
```

Impact: If an overflow occurs when increamenting `puppyRaffle::totalFees` by fees generated per raffle round an overflow might end up occuring and the `puppyRaffle::selectWinner` function call will always revert on every call with an overflow error causing a form of DoS where users can always enter raffle but no winners ever gets selected until the total fees is withdrawn to the fees address

Proof of Concept: Note: Due to a deeper problem in the `puppyRaffle::selectWinner` function where the fee is type casted to a `uint64` from a `uint256` fee data crucial to accont for actual fee value is lost so before running the proof of code below its important to resolve the bug due to type casting first.

PoC

```
1       function testOverFlowOfTotalFeesCausingDoSInSelectWinner() public {
2           // Enter maximum number of players to cause overflow
3           uint256 numOfPlayers = 100;
4           address[] memory players = new address[](numOfPlayers);
5           for (uint256 i = 0; i < players.length; i++) {
6               players[i] = address(uint160(i + 300));
7           }
8           puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
9           uint256 raffleBalance = address(puppyRaffle).balance;
10          console.log("Raffle Balance:", raffleBalance);
11
12          vm.warp(block.timestamp + duration + 1);
13          vm.roll(block.number + 1);
14
15          // Attempt to select winner
16          vm.expectRevert();
17          puppyRaffle.selectWinner();
18          uint256 totalFeess = puppyRaffle.totalFees();
19          address previousWinner = puppyRaffle.previousWinner();
20          uint256 previousWinnerBalance = previousWinner.balance;
21          console.log("Previous Winner:", previousWinner);
22          console.log("Previous Winner Balance:", previousWinnerBalance);
23          console.log("Total Fees after overflow attempt:", totalFeess);
24      }
```

Recommended Mitigation: The best approach to take in eliminating this bug and the underlying

bug deep within caused by typecasting is to change the `puppyRaffle::totalFees` variable to a `uint256` type and eliminating the need for type casting

```
contract PuppyRaffle is ERC721Enumerable, Ownable {

-   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.prevrandao))) % players.length;
        address winner = players[winnerIndex];
        // q: why not just do address(this).balance?
        uint256 totalAmountCollected = players.length * entranceFee;
        // q: is there a reason for prizePool being 80% and fee being 20%?
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;

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

        uint256 tokenId = totalSupply();
    }
}
```

Note: The above code is just a illustration of what to add and to remove and does not contain the full detail of the actual code.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` function returns `0` if the player is not active. This could be misleading, as it is also a valid index.

Description: If the player in the `PuppyRaffle::players` array is at index `0`, and `PuppyRaffle::getActivePlayerIndex` function returns `0` meaning a player is not active. This could be misleading, as it is also a valid index and the caller can assume that the player is active.

```
  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: This could be misleading, as it is also a valid index and the caller can assume that the player is active.

Informational

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

consider using specific versions of solidity in your contract instead of wide versions. For example instead of `pragma solidity ^0.8.13;` use `pragma solidity 0.8.13;`

[I-2] Using an outdated version of solidity is not recomended

solc frequently releases new solidty compiller versions. Using old versions prevents access to new solidity security checks. We recommend avoiding complex pragma statement.

[I-3] Check for address(0) when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 63

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 169

```
        feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` function does not follow the CHECKS-EFFECTS-INTERACTIONS pattern

The `PuppyRaffle::selectWinner` function does not follow the CHECKS-EFFECTS-INTERACTIONS pattern or CEI pattern.

```
-        (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 not recommended as its meaning is unclear

In the `PuppyRaffle::selectWinner` function, the magic number `80`, `20`, and `100` are used to calculate the fee and prize pool. The magic number is not recommended as its meaning is unclear.

```
+    uint8 private constant PRIZE_POOL_PERCENTAGE = 80;
+    uint8 private constant FEE_PERCENTAGE = 20;
+    uint8 private constant TOTAL_PERCENTAGE = 100;

-    uint256 prizePool = (totalAmountCollected * 80) / 100;
-    uint256 fee = (totalAmountCollected * 20) / 100;
+    uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
+    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / TOTAL_PERCENTAGE;
```

[I-6] `PuppyRaffle::_isActivePlayer` internal function is never used in the contract.

The `PuppyRaffle::_isActivePlayer` internal function is never used in the contract and this ends up being a dead code.

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

Gas

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

variables that are not subject to change later on should be declared as `constant` or `immutable`. This prevents the variable from being modified after it has been initialized, and prevents storage slots to be used unecessarily.

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 variable in a loop should be cached in memory

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