PuppyRaffle Audit Report

Version 1.0

Cyfrin.io

November 2, 2025

PuppyRaffle Audit Report

Dewangga Praxindo

Nov 2, 2025

Prepared by: Dewaxindo Lead Auditors:

- Dewangga Praxindo

Table of Contents

- Medium
  * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS vector, incrementing gas costs for future entrants
  * [M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
  * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  * [M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
- Gas
  * [G-1] Unchanged State Variables Should be Declared Constant or Immutable
  * [G-2] Storage Variables in a Loop Should be Cached
- INFO
  * [I-1] Unspecific Solidity Pragma
  * [I-2] Using Outdated Version of Solidity
  * [I-3] Address State Variable Set Without Checks
  * [I-4] Potentially erroneous active player index
  * [I-5] Magic Numbers
  * [I-6] Test Coverage
  * [I-7] Dead Code

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 DEWAXINDO 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

	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Table 1: Severity matrix by likelihood and impact

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Commit Hash: 0804be9b0fd17db9e2953e27e9de46585be870cf

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

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	0
Gas	2
Info	7
Total	16

Findings

HIGH

[H-1] Reentrancy Attack in PuppyRaffle::refund Allows Entrant To Drain Contract Balance

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

In the PuppyRaffle::refund function, the contract makes an external call to sendValue before updating the state variable. This means an attacker can create a malicious contract that calls refund in its fallback function, allowing them to recursively call refund multiple times before the state is updated.

The vulnerable code flow:

1. Contract checks that msg.sender is the player (Checks ✓)
2. Contract sends ETH to the attacker via sendValue (Interactions × - happens too early)
3. Attacker's fallback function calls refund again
4. Since state hasn't been updated (players[playerIndex] is still set), the checks pass again
5. Contract sends ETH again, and this repeats until the contract is drained
6. State is finally updated (players[playerIndex] = address(0)), but it's too late

Impact: An attacker can drain the entire contract balance by repeatedly calling refund through a malicious contract's fallback function. This is a critical vulnerability that can result in complete loss of funds for the protocol and all participants.

Proof of Concept:

See test/ProofOfCodes.t.sol for the full exploit:

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

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex); // First refund call
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex); // Recursive call before
↪   state update
        }
    }
}
```

The PoC demonstrates that an attacker can drain the entire contract balance, reducing it from the initial balance to zero.

Recommended Mitigation:

Follow the CEI (Checks, Effects, Interactions) pattern by updating state before making external calls:

```
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");

    // Effects: Update state FIRST
    players[playerIndex] = address(0);
    emit RaffleRefunded(playerAddress);

    // Interactions: External call LAST
    payable(msg.sender).sendValue(entranceFee);
}
```

Alternatively, use OpenZeppelin's `ReentrancyGuard`:

```
import {ReentrancyGuard} from
↪  "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract PuppyRaffle is ERC721, Ownable, ReentrancyGuard {
    function refund(uint256 playerIndex) public nonReentrant {
        // ... existing code ...
    }
}
```

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 109

```
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");

    // External call before state update - VULNERABLE
    payable(msg.sender).sendValue(entranceFee);

    // State update happens after external call
    players[playerIndex] = address(0);
    emit RaffleRefunded(playerAddress);
}
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

Description: Hashing `msg.sender`, `block.timestamp`, `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.

Impact: Any user can choose the winner of the raffle, winning the money and selecting the "rarest" puppy, essentially making it such that all puppies have the same rarity, since you can choose the puppy.

Proof of Concept:

There are a few attack vectors here.

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when / how to participate. See the solidity blog on prevrando here. `block.difficulty` was recently replaced with `prevrandao`.

2. Users can manipulate the `msg.sender` value to result in their index being the winner.

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

Recommended Mitigation: Consider using an oracle for your randomness like 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;
// myVar will be 18446744073709551615
myVar = myVar + 1;
// myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `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 first conclude a raffle of 4 players to collect some fees.
2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Proof Of Code

See `test/ProofOfCodes.t.sol` for the full exploit:

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
```

```
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a
        ↪  second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require
        ↪  check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
↪  active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- 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`.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS vector, 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 that the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

Additionally, this increased gas cost creates front-running opportunities where malicious users can front-run another raffle entrant's transaction, increasing its costs, so their enter transaction fails.

Impact: The impact is two-fold.

1. The gas costs for raffle entrants will greatly increase as more players enter the raffle.
2. Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails.

Proof of Concept:

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

- 1st 100 players: 6252039
- 2nd 100 players: 18067741

This is more than 3x as expensive for the second set of 100 players!

This is due to the for loop in the `PuppyRaffle::enterRaffle` function.

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

Proof Of Code

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

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

        // We will enter 5 players into the raffle
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // And see how much gas it cost to enter
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the 1st 100 players:", gasUsedFirst);

        // We will enter 5 more players into the raffle
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i + playersNum);
        }
        // And see how much more expensive it is
        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
        gasEnd = gasleft();
        uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the 2nd 100 players:", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
        // Logs:
        //     Gas cost of the 1st 100 players: 6252039
        //     Gas cost of the 2nd 100 players: 18067741
}
```

Recommended Mitigation: There are a few recommended mitigations.

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 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");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
    function withdrawFees() external {
@>      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: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
    function withdrawFees() external {
-       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");
    }
```

[M-3] 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. There 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-4] 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 owness on the winner to claim their prize. (Recommended)

Gas

[G-1] Unchanged State Variables Should be Declared Constant or Immutable

Description: 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

Description: Everytime you call `players.length` you read from storage, as opposed to memory which is more gas efficient.

```
+     uint256 playersLength = players.length;
-     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");
            }
        }
```

INFO

[I-1] Unspecific Solidity Pragma

Description: 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 Outdated Version of Solidity

Description: Using outdated version is not recommended. Consider using latest stable version.

[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: 67

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 193

```
        feeAddress = newFeeAddress;
```

[I-4] Potentially erroneous active player index

Description: The `getActivePlayerIndex` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return 2**256-1 (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

[I-5] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called “magic numbers”.

Recommended Mitigation: Replace all magic numbers with constants.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public 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] Test Coverage

Description: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

```
| File                                | % Lines        | % Statements   | % Branc
| ----------------------------------- | -------------- | -------------
- | -------------- | ------------- |
| script/DeployPuppyRaffle.sol        | 0.00% (0/3)    | 0.00% (0/4)    | 100.00%
| src/PuppyRaffle.sol                 | 82.46% (47/57) | 83.75% (67/80) | 66.67%
| test/auditTests/ProofOfCodes.t.sol | 100.00% (7/7)  | 100.00% (8/8)  | 50.00%
| Total                               | 80.60% (54/67) | 81.52% (75/92) | 65.62%
```

Recommended Mitigation: Increase test coverage to 90% or higher, especially for the `Branches` column.

[I-7] Dead Code

Functions that are not used. Consider removing them.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 199

```
    function _isActivePlayer() internal view returns (bool) {
```