2. PuppyRaffle Audit

TOTAL FINDINGS (17)

- **HIGH (2)**
- **MEDIUM (3)**
- **LOW (2)**
- **INFORMATIONAL (10)**

HIGH SEVERITY FINDINGS (2)

[S-H] Mishandling of ETH allows an attacker to break the contract resulting in fees being stuck forever.

Description

The contract function mishandle ETH in the function `PuppyRaffle::withdrawFees` when it performs the `require` check that `balance == totalFees`.

This line can break the contract when ETH is forced into the contract as the internal accounting is not able to account for ETH being received due to a selfdestruct call made on another contract.

ETH can be forced into the contract when another contract selfdestrcuts and forces its ETH into the specific address (ie. this contract). This results in the balance being different to the value of totalFees and therefore the code fails and reverts meaning the fees are stuck forever.

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

Impact

Fees are permanently stuck in the contract and cannot be retrieved by anyone.

Proof of Concept

Deploying the `AttackPuppyRaffle` contract below with an ETH payment and then calling the function `AttackPuppyRaffle::attack` will force the ETH of the `AttackPuppyRaffle` contract into the `PuppyRaffle` contract address making the balance greater than the internal accounting expects. This causes the `PuppyRaffle::withdrawFees` function to fail on the check: `require(address(this).balance == uint256(totalFees)` resulting in the fees being stuck forever.

See Test case: `PuppyRaffleMishandlingEthTest::test_mishandling_eth` for full PoC.

```
contract AttackPuppyRaffle {
    PuppyRaffle target;

    constructor(PuppyRaffle _target) payable {
        target = _target;
    }

    function attack() external payable {
        selfdestruct(payable(address(target)));
    }
}
```

Recommended mitigation

Change the operator in the check to be `>=` so if the contract balance is greater than then internal accounting expects the fees can still be withdrawn. Alternatively remove the check from the code and perform the accounting checks using a safer mechanism that still enables fees to be withdrawn.

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

[S-H] Reentrancy vulnerability allows an attacker to drain/steal all of the entrance fees from the contract.

Description

Reentrancy attack allows attacker to drain all the entrance fees of the entire contract. This occurs in the `PuppyRaffle::refund` function, shown below.

```
    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);
    }
```

Impact

The entire protocol is at risk as an attacker can steal all of the entrance fees from the contract.

Proof of Concept

See: `PuppyRaffleReentrancyTest::test_reentrancy`

The output of the test demonstrates that the attacker can drain the entire contract balance by issuing reentrant calls to the `PuppyRaffle::refund` function. In the test we enter 10 victim players and then we enter the attacker player who immediately issues a refund, stealing all the victims funds (10 ether) and reclaiming their own entraceFee (1 ether).

```
Ran 1 test for
test/PuppyRaffleReentrancyTest.t.sol:PuppyRaffleReentrancyTest
[PASS] test_reentrancy() (gas: 289321)
Logs:
  Attacker before attack balance =  1000000000000000000
  PuppyRaf before attack balance =  10000000000000000000
  Attacker after attack balance =  11000000000000000000
  PuppyRaf after attack balance =  0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.56ms (1.12ms
CPU time)

Ran 1 test suite in 443.31ms (2.56ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

Recommended mitigation

Move the call to `sendValue` to the last line of the contract, and change the call to a safe call and check the status of the return value to ensure it succeeded without reverting.

MEDIUM SEVERITY FINDINGS (3)

[S-M] A DOS due to ever-increasing GAS fees results in a limited number of players entering the raffle, reducing the amount of fees able to be collected.

Description

Denial Of Service (DOS) via ever increasing gas fees. Eventually the GAS fees will increase to a level that prevents new players from entering the raffle. This is due to the length of the players array constantly growing with each subsequent player entry in the `PuppyRaffle::enterRaffle` function.

Eg.

```
Gas used for 1000 players  = 417422121
417422121 * 0.000 000 001  =  0.417 422 121 (0.417422121 ETH) ~ 1500 USD

Gas used for 5000 players = 9.977 312 637 (9.9 ETH) ~ 30,000 USD

Gas used for 10000 players = 39 680 291 015 (39.6 ETH) ~ 118,000 USD
```

Impact

Artificially limits the number of players who would enter the raddle due to the entry cost.

Proof of Concept

See: `PuppyRaffleStatefulAuditTest::test_gasDOS`

Recommended mitigation

A mapping might be a better option for the duplicate check. Alternatively the duplicate check may be able to be removed as a player can simply enter multiple times with multiple wallets.

[S-M] Function `PuppyRaffle::getActivePlayerIndex` returns zero when player is not found but also will zero for the first player in the players array.

Description

The function returns zero if the player is not found in the players array. This means sending a non-active address returns zero which corresponds to the first player in the players array, that is, the player at index zero.

```
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

The function incorrectly returns zero, which is the index of the first player for an address which does not appear in the array.

Proof of Concept

See: `PuppyRaffleStatefulAuditTest::test_playerIndexZeroReturnedWhenPlayerNotInPlayersArray`

```
function test_playerIndexZeroReturnedWhenPlayerNotInPlayersArray() public {
    address[] memory players = new address[](1);
    players[0] = playerOne;

    uint256 index = puppyRaffle.getActivePlayerIndex(playerOne);
    assertEq(index, 0);

    uint256 notFoundIndex = puppyRaffle.getActivePlayerIndex(notPlayer);
    assertEq(notFoundIndex, 0);
}
```

```
[PASS] test_playerIndexZeroReturnedWhenPlayerNotInPlayersArray() (gas:
16834)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.40ms
(126.94µs CPU time)
```

Recommended mitigation

Change the function to `return -1` when player not found in the players array.

[S-M] Underflow occurs when casting a uint265 to a uint64 resulting in a loss of precision

Description
The function `PuppyRaffle::selectWinner` casts a uint256 to a uint64 resulting in a significant loss of fees.

```
    uint256 fee = (totalAmountCollected * 20) / 100;
@>  totalFees = totalFees + uint64(fee);
```

Impact

Fees lost.

Proof of Concept
See: `PuppyRaffleStatefulAuditTest::test_precision`

```
Ran 1 test for
test/PuppyRaffleStatefulAuditTest.t.sol:PuppyRaffleStatefulAuditTest
[PASS] test_precision() (gas: 4292)
Logs:
  Fee is of type uint256 =
115792089237316195423570985008687907853269984665640564039457584007913129639935

  Fee when cast to uint64 =  18446744073709551615
```

Recommended mitigation

- Use consistent types when performing calculations.

- Use a current version of solidity >= 0.8.20 so an error would be thrown at runtime.

LOW SEVERITY FINDINGS (2)

[S-L] Weak Cryptographic random number generator used to choose Raffle winner meaning Raffle winner could be manipulated by validator nodes with control over block parameters.

Description

Weak cryptographic random number used to choose winner could allow a malicous validator to manipulate the result of the Raffle.

Impact

Raffle winner could be manipulated by validators.

Proof of Concept

N/A

Recommended mitigation

Use a purpose built, trusted and well tested library for random number generation.

[S-L] Weak Cryptographic Randomness used to determine minted puppy rarity could enable an attacker to mint a rare puppy rather than get a random rarity.

Description

Weak cryptographic randomness used to determine rarity can allow a malicious validator to mint the type of puppy they want rather than a randomly generated puppy.

Impact

Attacker may be able to manipulate the rarity of the minted Puppy.

Proof of Concept

N/A

Recommended mitigation

Use a purpose built, trusted and well tested library for generating random numbers.

INFORMATIONAL/GAS FINDINGS (10)

[S-I] Floating pragma allows different versions of solidity to be used which can lead to variations in behaviour in different environments making debugging harder.

Description

A floating pragma is in use which can lead to variations in behaviour when code is executed in different environment.

Impact

Using a floating solidity version can result in inconsistent behaviour depending on version in use. This can make it very difficult to trace and track bugs or potentially for profiling GAS costs etc.

Recommended mitigation

Pin the exact version requried to be used. Should be the current stable version, eg:

```
pragma solidity 0.8.20;
```

[S-I] Outdated SolC version in use results in lack of boundry checking for integer overflow/underflow.

Description

The compiler version is not up-to-date meaning that improvements in security and optimisations provided by current versions are not in use when the contract is compiled.

Impact

Boundry checks for integer overflow and underflow are not performed at runtime meaning precision can be lost and/or numbers can wrap around exceeding expected bounds. Under some circumstances very large values may be used where much smaller values are expected, for example, when withdrawing funds a user may be able to overflow an integer and withdraw 10x or 100x the expected amount.

Recommended mitigation

Use a newer/current version of solc. Eg. `pragma solidity 0.8.20;`

[S-I] The Raffle Duration variable `PuppyRaffle.raffleDuration`, should be an immutable variable signaling to the compiler that it should not be changed once it's been set.

Description

The raffleDuration variable is not marked as immutable even though it's not intended to be changed after being set. This is inconsistent with the other immutable variables and reduces the quality of code by removing compiler safety checking.

```
uint256 public raffleDuration;
```

Impact

Unintentional changes are possible by developer error, code quality and readabilty is reduced.

Recommended mitigation
Add immutable modified to improve code quality and consistency and catch unintentional value changes.

[S-I] Inconsistent use of data types which results in loss of precision and potential calculation errors and boundry overflow or underflow errors.

Description

Using mixed variable types can result in mathematical errors being introduced in calculations.
`uint64 public totalFees = 0;`

Impact

Using different data types in calculations requires conversions and casts which can introduce subtle errors and boundry violations such as integer under/overlow.

Recommended mitigation

Use the same type for variables that are used within calculations when possible.

[S-I] The `PuppyRaffle::enterRaffle` function for-loops repeatedly read from `array.length` rather than memory increasing GAS costs.

Description

Continued calls to array.length increases the GAS fees.

```
+   uint256 newPlayersLength = newPlayers.length
+   for (uint256 i = 0; i < newPlayersLength; i++) {
-   for (uint256 i = 0; i < newPlayers.length; i++) {
        players.push(newPlayers[i]);
    }


+   uint256 playersLength = players.length
+   for (uint256 i = 0; i < playersLength - 1; i++) {
-   for (uint256 i = 0; i < players.length - 1; i++) {
+       for (uint256 j = i + 1; j < playersLength; j++) {
-       for (uint256 j = i + 1; j < players.length; j++) {
            require(players[i] != players[j], "PuppyRaffle: Duplicate
player");
        }
    }
```

Impact

Wasted GAS fees when calling the `enterRaffle` function due to repeated storage reads.

Recommended mitigation

- Cache array lengths so they're not continually called in for-loops.
- Read from memory rather than from storage variables when possible.

[S-I] The `PuppyRaffle::RaffleEnter` event is emitted even if the `newPlayers` array is empty which results in a false positive when observing emitted events

Description

The event `PuppyRaffle::RaffleEnter(newPlayers)` is emitted when the newPlayers array is empty.

Impact

RaffleEnter events are emitted when no players are entered.

Recommended mitigation

Only emit the event when the newPlayers array has a length greater than zero.

[S-I] Magic Numbers used for calculations making the code harder to read and maintain

Description

```
Magic numbers are used in calculations within the `selectWinner` function.
These should be changed to constants to make code easier to read and
maintain.
```

Impact
Code is more difficult to read and maintain.

Recommended mitigation

Use constant values with meaningful variable names.

[S-I] Data is stored in the `PuppyRaffle::previousWinner` storage variable but does not appear to be used, leading to increased GAS costs and code complexity.

Description

The variable previousWinner does not appear to be used used.

```
previousWinner = winner;
```

Impact

Wasted GAS fees and increased code complexity.

Recommended mitigation

Remove unused variables from code base.

[S-I] The function `PuppyRaffle::changeFeeAddress` fails to check for the zero address, which could lead to fees being mistakenly burned.

Description

The `PuppyRaffle::changeFeeAddress` function does not guard against setting the address(0) which could result in lost fees if set incorrectly.

```
function changeFeeAddress(address newFeeAddress) external onlyOwner {
    feeAddress = newFeeAddress;
    emit FeeAddressChanged(newFeeAddress);
}
```

Impact

Potential for lost fees due to human error.

Recommended mitigation

Perform a check for the zero address.

[S-I] The function `PuppyRaffle::_isActivePlayer` is not used and could be removed to save GAS costs.

Description

// The function is not used and the code could be removed.

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

Impact

Unused code increases code complexity and GAS costs.

Recommended mitigation

Remove unused code