Protocol Audit Report

Abdikadr Jerow

September 13, 2024

Protocol Audit Report

Version 1.0

Cyfrin.io

September 15, 2024

Protocol Audit Report

Abdikadr Jerow

September 13, 2024

Prepared by: Cyfrin Lead Auditors: Abdikadr Jerow

Smart Contract Audit Report: PuppyRaffle

Table of Contents:

Disclaimer

Abdikadr Jerow 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 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
Likelihood	High	H	H/M	M
	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

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Protocol Summary

The protocol under review is designed to perform a raffle mechanism called “PuppyRaffle” through smart contracts. Its primary functionality includes allowing users to enter a raffle by sending funds, selecting a winner randomly, and handling refunds for users who wish to exit before the raffle concludes. The raffle operates on-chain, with key elements such as random number generation for selecting a winner, participant tracking, and fee management.

Several vulnerabilities and inefficiencies were identified in the protocol that could potentially impact its security and operational efficiency. Critical areas for improvement include securing randomness generation, optimizing the refund process, and reducing gas costs through code optimizations.

Key features:

Raffle participation via smart contracts. Random winner selection based on on-chain variables. Fee management system.

Issues found

Severity	Number of issues found
High	1
Medium	2
Low	2
Info	5
Total	10

Findings

High Severity Bugs

[S-1] Lack of Randomness Security in `selectWinner()`

Description: The contract relies on the hash of on-chain variables such as msg.sender, block.timestamp, and block. difficulty to generate randomness in the selectWinner() function. These values can be manipulated or predicted by miners, leading to potential abuse of the randomness mechanism.

Impact: Attackers (such as miners) can influence the outcome of the raffle, giving them a higher chance of winning.

Proof of Concept:

```
uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timest
```

Test case for Proof:

```
it("Test manipulation of randomness by mining node", async function () {
  const initialBlock = await ethers.provider.getBlockNumber();
  await raffle.enterRaffle([addr1.address, addr2.address, addr3.address], {
    value: entranceFee.mul(3),
  });
  await ethers.provider.send("evm_setBlockTimestamp", [initialBlock + 100]); //

  // Now simulate block mining with manipulated difficulty
  await ethers.provider.send("evm_mine", [difficultyManipulation]);

  await raffle.selectWinner();
  const winner = await raffle.previousWinner();

  // Assert the manipulated winner
  assert.equal(winner, addr1.address); // addr1 rigged the raffle.
});
```

Recommended Mitigation: Use Chainlink VRF or another trusted randomness oracle to generate truly unpredictable random numbers.

Medium Severity Bugs

[S-2] Refund Mechanism Leaves Empty Slots in `players`

Description: The refund() function does not remove refunded players from the players array but instead sets their address to address(0). This results in empty spots in the players array, which could cause confusion or inefficiencies when selecting a winner.

Impact: Players are not fully removed from the raffle after a refund, leading to inefficiencies when calculating the winner and potentially increasing gas costs.

Proof of Concept:

```
players[playerIndex] = address(0);
```

Test case for Proof:

```
it("Test refund leaves empty spot in players array", async function () {
  await raffle.enterRaffle([addr1.address, addr2.address], {
    value: entranceFee.mul(2),
  });

  await raffle.refund(0, { from: addr1.address });
  const playerAtIndex0 = await raffle.players(0);

  assert.equal(playerAtIndex0, ethers.constants.AddressZero); // Ensures addr1 w
});
```

Recommended Mitigation: Consider removing the refunded player entirely from the players array by shifting the array or implementing a more gas-efficient mechanism, such as a linked list.

[S-3] Lack of Input Validation for `changeFeeAddress()`

Description: The changeFeeAddress() function allows the contract owner to set the feeAddress to any address, including the zero address. Setting it to address(0) would break the fee withdrawal functionality.

Impact: If the owner mistakenly sets the feeAddress to address(0), fees will be locked in the contract permanently.

Proof of Concept:

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

Test case for Proof:

```
it("Test setting feeAddress to address(0)", async function () {
  await raffle.changeFeeAddress(ethers.constants.AddressZero);

  await expect(raffle.withdrawFees()).to.be.revertedWith(
    "PuppyRaffle: Failed to withdraw fees"
  ); // FeeAddress is zero.
});
```

Recommended Mitigation: Add input validation to ensure that newFeeAddress is not the zero address:

```
require(newFeeAddress != address(0), "PuppyRaffle: Invalid fee address");
```

Low Severity Bugs

[S-6] Inefficient Use of Loops in `enterRaffle()`

Description: In enterRaffle(), nested loops are used to check for duplicate participants, which can be inefficient as the number of participants grows.

Impact: This could result in higher gas costs for large numbers of participants.

Proof of Concept:

```
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");
    }
}
```

Test case for Proof:

```
it("Test inefficiency for large input sizes", async function () {
  const participants = [...Array(100).keys()].map((i) => addr1.address); // 100
  await expect(raffle.enterRaffle(participants)).to.be.revertedWith(
    "PuppyRaffle: Duplicate player"
  );

  const gasUsed = await raffle.estimateGas.enterRaffle([addr1.address]);
  console.log("Gas used: ", gasUsed.toString());
});
```

Recommended Mitigation: Consider using a mapping to track participants and prevent duplicates in O(1) time.

[S-9] Redundant Initialization of `raffleStartTime`

Description: The raffleStartTime is initialized twice: once in the constructor and again after the selectWinner() function. This is redundant.

Impact: Increased gas costs, though minor, during contract deployment.

Proof of Concept:

```
raffleStartTime = block.timestamp;
```

Test case for Proof:

```
it("Test redundant initialization of raffleStartTime", async function () {
  const startTime = await raffle.raffleStartTime();
  await raffle.selectWinner();
  const newStartTime = await raffle.raffleStartTime();

  assert.notEqual(startTime, newStartTime); // Redundant update detected.
});
```

Recommended Mitigation: Remove the redundant initialization of raffleStartTime from the constructor.

Informational Bugs

[S-4] Inefficient Gas Usage in `getActivePlayerIndex()`

Description: The getActivePlayerIndex() function loops through the players array to find a player's index, which can be gas inefficient.

Impact: Potential increase in gas costs for large numbers of players.

Proof of Concept:

```
for (uint256 i = 0; i < players.length; i++) {
    if (players[i] == player) {
        return i;
    }
}
```

Recommended Mitigation: Consider using a mapping of player addresses to their index to optimize lookups.

[S-5] Potential Gas Waste with `sendValue()` in Refund

Description: The refund() function uses sendValue() to transfer funds, which might not be the most gas-efficient method for simple refunds.

Impact: Slightly higher gas costs for refund operations.

Proof of Concept:

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

Recommended Mitigation: Use a simple transfer() if the transfer value is small, as it has lower overhead.

[S-7] Storage Packing for `feeAddress` and `totalFees`

Description: The feeAddress and totalFees variables are not packed in storage efficiently.

Impact: Increased gas usage for storage reads/writes.

Proof of Concept:

```
address public feeAddress;
uint64 public totalFees = 0;
```

Recommended Mitigation: Pack the variables into the same storage slot to optimize gas usage.

[S-8] Use of `require()` for Basic Checks

Description: require() is used for basic checks that might be more gas-efficient using assert().

Impact: Increased gas usage for contract execution.

Proof of Concept:

```

solidity
require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send en
```

Recommended Mitigation: Review require() statements and replace with assert() where the condition should never fail.

[S-10] Hardcoded IPFS URIs in Contract

Description: Hardcoded IPFS URIs for metadata increase the contract's bytecode size and deployment costs.

Impact: Increased gas costs during deployment.

Proof of Concept:

```
string private commonImageUri = "ipfs://QmSsYRx3LpDAb1GZQm7zZ1AuHZjfbPkD6J7s9r41
```

Recommended Mitigation: Consider storing URIs off-chain or using a more flexible on-chain structure.

Conclusion

This audit identified multiple issues, ranging from high to informational severity. The high-severity bug, in particular, requires urgent attention to prevent manipulation of randomness in the raffle. Several optimizations can also be made to reduce gas consumption and improve the overall efficiency of the contract.