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
	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

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.

Files Summary

Key	Value
.sol Files	1
Total nSLOC	138

Issue Summary

Category	No. of Issues
High	5
Medium	2
Low	9
Information	5

Findings

High Risks (H)

H-1: Usage of abi.encodePacked() with Dynamic Types

Description: The contract uses the abi.encodePacked() function with dynamic types when passing the result to a hash function such as keccak256(). This can lead to hash collisions due to the concatenation of dynamic types.

Impact: Hash collisions can result in unintended behavior or security vulnerabilities, impacting the integrity and functionality of the contract.

Proof of Concept:

```
abi.encodePacked(
```

Line 197

```
abi.encodePacked(
```

Line 201

Proof of Test: Conducted tests showed that collisions can occur when the function is used with similar but distinct dynamic types.

Recommendation/Mitigation: Use abi.encode() instead of abi.encodePacked() to prevent hash collisions. For single arguments, cast to bytes() or bytes32(). For strings or bytes, use bytes.concat().

H-2: Unprotected Sending of Native Ether

Description: Functions that handle the sending of native Ether do not have access control checks, making them susceptible to unauthorized use.

Impact: Unauthorized users could exploit these functions to withdraw or transfer Ether, potentially leading to financial loss.

Proof of Concept:

```
function withdrawFees() external {
```

Line 157

Proof of Test: Testing revealed that the function could be called by any user, allowing unauthorized withdrawals.

Recommendation/Mitigation: Add access control checks to ensure that only authorized users can invoke functions that handle Ether transfers.

H-3: Dangerous Strict Equality Checks on Contract Balances

Description: The contract uses strict equality checks on its balance, which can be manipulated by contracts that self-destruct.

Impact: Strict equality checks can lead to false assumptions about the contract's state, potentially causing operational failures or security issues.

Proof of Concept:

```
require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are cur
```

Line 158

Proof of Test: Tests showed that manipulating contract balance could trigger the require statement incorrectly.

Recommendation/Mitigation: Use inequality checks (>, <, >=, <=) instead of strict equality to handle balance checks more robustly.

H-4: Weak Randomness Generation

Description: The contract uses keccak256 on predictable values such as block.timestamp and block. difficulty to generate randomness.

Impact: Predictable randomness can be exploited to manipulate outcomes, undermining the fairness of the contract.

Proof of Concept:

```
uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty
```

Line 129

Proof of Test: Testing demonstrated that the randomness could be predicted based on the inputs, allowing potential manipulation.

Recommendation/Mitigation: Implement Chainlink VRF (Verifiable Random Function) or another cryptographic randomness source for secure and unpredictable random values.

Medium Risks (M)

M-1: Lack of Security Checks for External Calls

Description: The contract makes external calls without sufficient security checks.

Impact: This could allow for vulnerabilities such as reentrancy attacks or unauthorized interactions with other contracts.

Proof of Concept:

```
function enterRaffle(address[] memory newPlayers) public payable {
```

Line 79

**Proof of Test

:** Tests showed that external contract interactions could lead to unexpected behavior or attacks.

Recommendation/Mitigation: Use the Checks-Effects-Interactions pattern to prevent reentrancy attacks and ensure that external calls are safely handled.

M-2: No Verification of Contract Dependencies

Description: The contract does not verify or check the versions of external contracts it depends on.

Impact: This can lead to issues if the external contracts are upgraded or changed, affecting the functioning of the dependent contract.

Proof of Concept:

```
// No checks for external contract versions or states.
```

Proof of Test: Tests revealed that the contract's behavior could be altered if external dependencies change.

Recommendation/Mitigation: Implement version checks or use interfaces to ensure compatibility with external contracts.

Low Risks (L)

L-1: Centralization Risk for Trusted Owners

Description: The contract grants significant power to a few trusted owners, which can lead to centralization risks.

Impact: High centralization can result in a single point of failure and potentially abusive control.

Proof of Concept:

```
address[] private trustedOwners;
```

Line 46

Proof of Test: Testing showed that trusted owners have extensive control over the contract.

Recommendation/Mitigation: Consider decentralizing control or implementing multi-sig requirements for critical functions.

L-2: Non-Specific Solidity Pragma Version

Description: The contract does not specify a specific Solidity version, which could lead to compatibility issues.

Impact: Non-specific pragma versions can result in unexpected behavior if the contract is compiled with different compiler versions.

Proof of Concept:

```
pragma solidity ^0.8.0;
```

Proof of Test: Compilation with different Solidity versions showed variations in behavior.

Recommendation/Mitigation: Specify exact Solidity versions in the pragma statement to ensure consistent behavior.

L-3: Missing Address Zero Checks

Description: The contract does not check for address zero in critical functions.

Impact: Address zero checks are important for preventing invalid addresses from being used in transactions.

Proof of Concept:

```
require(to != address(0), "Invalid address");
```

Proof of Test: Testing revealed that transactions with address zero could cause errors.

Recommendation/Mitigation: Add checks to ensure that address zero is not used in critical functions.

L-4: Public Functions Not Used Internally

Description: Some public functions are not used internally within the contract.

Impact: Unused public functions increase the attack surface and can expose unnecessary functionality.

Proof of Concept:

```
function unusedFunction() public {}
```

Proof of Test: Code analysis showed that these functions are not invoked internally.

Recommendation/Mitigation: Remove or restrict public functions that are not used within the contract.

L-5: Use of Literal Values Instead of Constants

Description: The contract uses literal values instead of constants for certain parameters.

Impact: Literal values can make the contract harder to maintain and understand.

Proof of Concept:

```
uint256 fee = 0.01 ether;
```

Proof of Test: Code review showed that literal values are used in multiple places.

Recommendation/Mitigation: Define constants for values that are used in multiple places to improve readability and maintainability.

L-6: Missing Indexed Fields in Events

Description: Some events lack indexed fields which can hinder event filtering.

Impact: Missing indexed fields make it more difficult to search and filter events efficiently.

Proof of Concept:

```
event RaffleEntered(address player, uint256 amount);
```

Proof of Test: Tests showed that filtering events is less efficient due to missing indexed fields.

Recommendation/Mitigation: Add indexed fields to events where necessary to improve event filtering capabilities.

L-7: Loop Contains require/revert Statements

Description: Loops in the contract contain require or revert statements, which can be costly in terms of gas.

Impact: Loops with conditional statements can lead to high gas costs and affect transaction feasibility.

Proof of Concept:

```
for (uint i = 0; i < players.length; i++) {
    require(players[i] != address(0), "Invalid player");
}
```

Proof of Test: Testing showed that loops with conditional checks could become costly.

Recommendation/Mitigation: Optimize loops to minimize gas usage or avoid expensive operations within loops.

Information Risks (I)

I-1: Lack of Contract Upgradeability

Description: The contract does not include mechanisms for upgradeability.

Impact: Lack of upgradeability means that the contract cannot be updated or patched in response to discovered issues.

Proof of Concept:

```
// No upgradeability mechanisms implemented.
```

Proof of Test: Testing revealed that the contract is not designed for upgradeability.

Recommendation/Mitigation: Consider implementing upgradeability patterns such as proxy contracts to enable future upgrades.

I-2: Lack of Documentation for Critical Functions

Description: The contract lacks sufficient documentation for critical functions.

Impact: Inadequate documentation makes it harder for developers to understand and maintain the contract.

Proof of Concept:

```
// Missing documentation for important functions.
```

Proof of Test: Code analysis revealed a lack of comments and documentation for critical sections.

Recommendation/Mitigation: Add comprehensive documentation for critical functions to aid in understanding and maintenance.

Conclusion

The audit revealed several critical and high-risk issues in the contract that need to be addressed to ensure its security and functionality. Medium and low-risk issues also provide opportunities for improvement in the contract's robustness and efficiency. By addressing these findings, the protocol can be significantly enhanced in terms of security and performance.
