Prepared by: Idriis Perrin Lead Auditor:

- Idriis Perrin

Table of Contents

- [G-2] Storage variables in a loop should be cached.
    - Informationional
- [I-1]: Solidity pragma should be specific, not wide
    - [I-2]: Current version of solidity is outdated.
    - [I-3] `PuppyRaffle::selectWinner` should follow CEI.
    - [I-4] `PuppyRaffle::_isActivePlayer` is never used and shold 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:
    - `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 YOUR_NAME_HERE 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: e30d199697bbc822b646d76533b66b7d529b8ef5
- 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.

Issues found

Severity	Number of issues found
High	2
Gas	2
Low	3
info	4
Total	14

Findings

High

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

Description: The `PuppyRaffle::refund` does not follow CEI (Checks, Effects, Interactions) and as a result enables attackers to drain the contract balance.

In the `PuppyRaffle::refund` function we first make an external call to msg.sender and only after we make the 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");

        // @audit Reentrancy
        payable(msg.sender).sendValue(entranceFee);

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

A player who entered the raffle could have a `fallback/recieve` function that calls the `PuppyRaffle::refund`function again and claim multiple refunds until the account is drained.

Impact: All funds can be stolen by the attacker.

Proof of concept:

6. The `refund` function allows a player to refund their entrance fee. 2. The function sends the entrance fee back to the player. 3. The function then sets the player's address in the `players` array to `address(0)`. 4. The function emits a `RaffleRefunded` event. 5. The function is not marked as `nonReentrant`

Recomended mitigation:

To prevent this, we should have the `PuppyRaffle::refund` function update the `players` array before making the external call.

[H-2] Weak Randomness in `PuppyRaffle::seleectwinner` allows users to influence or predict the winner.

Description: Hashing `msg.sender`, `block.timestamp`, `block.difficulty` together creates a predictible result. A predictible number is not good randomness. Malicious users can manipulate these values or know them ahead of time.

Impact: Any user can influence the winner of the raffle.

Proof of concept:

User can manipulate their `msg.sender` value to result in the address being used to generate the winner of the raffle.

Recomended mitigation: Use a cryptographically secure random number generator such as chainlink VRF to generate a random number

Low

[L-1] `PuppyRaffle::getactivePlayersIndex()` reuens 0 for nonexistent players and players at index 0, causing all players to think they have not entered the raffle.

Description:

```
function getActivePlayerIndex(
    address player
) external view returns (uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
    return 0;
}
```

Impact A player at index 0 will think they have not entered the raffle. and attept to enter again which will waste gas.

Proof of concept:

7. User enters the raffle, they are the first person

8. `PuppyRaffle::getactiveplayerIndex` returns 0 User thinks they have not entered the raffle

Recomended mitigation:

Revert if the player is not in the array instead of returning 0.

Gas

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

Reading from storage is more expensive than reading from a constant or immutable variable.

Instances `Puppyraffle::commonImageUri` Should be `constant` `Puppyraffle::legendaryImageUri` should be `constant` `Puppyraffle::rareImageUri` should be `constant` `Puppyraffle::raffleDuration` should be `immutable`

[G-2] Storage variables in a loop should be cached.

When `players.length` is called you must read from storage which is more gas expensive than memory.

```
        +    uint256     playerlength = players.length
        -    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"
                 );
             }
 }


                 ```
```

```
### Informationional
## [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;`
```
```

- Found in src/PuppyRaffle.sol [Line: 2](src/PuppyRaffle.sol#L2)

	```solidity
	pragma solidity ^0.7.6;
	```

[I-2]: Current version of solidity is outdated.

Recomendateion
Newer version recomended `0.8.18`

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.

[I-3] `PuppyRaffle::selectWinner` should follow CEI.

[I-4] `PuppyRaffle::_isActivePlayer` is never used and shold be removed.