Protocol Audit Report

Version 1.0

xByteNeo

August 29, 2025

Puppy Raffle Audit Report

xByteNeo

August 25, 2025

Prepared by: xByteNeo

Lead Security Researcher:

- @xByteNeo

Table of Contents

    * [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
  - Medium
    * [M-1] Looping thorough players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potencial denial of service (DoS) attack, incrementing gas costs for future entrants
    * [M-2] Balance validation in `PuppyRaffle::withdrawFees` enables denial of service through forced ETH transfers
    * [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  - Informational
    * [I-1] Floating pragmas
    * [I-2] No validation for zero length players
    * [I-3] Missing checks for `address(0)` when assigning values to address state variables
    * [I-4] Magic Numbers
  - Gas
    * [G-1] Unchanged state variables should be declared constant or immutable

Protocol Summary

Puppy Raffle is a protocol for raffling off puppy NFTs with varying rarities. When users enter the raffle, a portion of their entrance fees goes to the eventual winner, while another portion is collected as protocol fees and sent to a designated fee address that can be updated by the protocol owner.

Disclaimer

The xByteNeo solo auditor 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 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

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

Scope

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

- Solc version: ^0.7.6
- Chain(s) to deploy to; Ethereum

Roles

- Owner: Deployer of the protocol
  (a) can change the wallet address to which fees are sent through the changeFeeAddress function.
- Player: Participant of the raffle
  (a) can call enterRaffle function
  (b) can refund value through refund function.

Executive Summary

The PuppyRaffle contract audit revealed multiple critical security vulnerabilities that pose significant risks to user funds and protocol integrity. The contract suffers from fundamental issues including reentrancy attacks, lack of input validation, predictable randomness, arithmetic overflow/underflow vulnerabilities, and denial of service attack vectors. These vulnerabilities could lead to complete drainage of contract funds, manipulation of raffle outcomes, and prevention of normal protocol operations. We strongly recommend addressing all identified issues before deploying to mainnet, with particular focus on the high-severity vulnerabilities that directly threaten user funds.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	0
Informational	2
Total	9

During the audit, a total of 9 issues were identified: - 4 High severity issues: lack of input validation allowing fund loss, reentrancy vulnerability enabling fund drainage, predictable randomness compromising raffle fairness, and arithmetic overflow/underflow risks. - 3 Medium severity issues: denial of service attack through gas cost manipulation, forced ETH transfer DoS vulnerability, and unsafe type casting in fee calculations. - 2 Informational issues: floating pragma versions and magic numbers usage.

The High severity issues pose immediate threats to user funds and protocol integrity, with the reentrancy vulnerability being particularly critical as it could lead to complete contract drainage. The Medium severity issues create operational risks and potential DoS scenarios that could disrupt normal protocol functionality.

Findings

High

[H-1] Reentrancy attack on `PuppyRaffle::refund` allowing drain of funds

Description: The `PuppyRaffle::refund` function updates the players array after making an external call. This creates a significant window for reentrancy attacks, where an attacker can call the `PuppyRaffle::refund` function multiple times and steal the contract's funds.

```
    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 the player is refunded before the players array is
            updated opening the reentrancy window
        payable(msg.sender).sendValue(entranceFee);

        // @audit the players array is updated after external
        players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

Impact: Attacker can call the `PuppyRaffle::refund` function multiple times, and steel the contract funds.

Proof of Concept: Include the test in the `PuppyRaffleTest.t.sol` file and run it with `forge test --mt test_reentrancyRefund -vvv` to check the impact fisibility.

PoC

```
    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 attacker = new ReentrancyAttacker(
            puppyRaffle);
        address attackerAddress = makeAddr("attacker");
        vm.deal(attackerAddress, 1 ether);

        uint256 startingAttackerContractBalance = address(attacker).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

        vm.prank(attackerAddress);
        attacker.attack{value: entranceFee}();

        uint256 endingAttackerContractBalance = address(attacker).balance;
        uint256 endingPuppyRaffleBalance = address(puppyRaffle).balance;

        console.log("Starting attacker contract balance: ", startingAttackerContractBalance);
        console.log("Starting puppy raffle balance: ", startingPuppyRaffleBalance);
        console.log("Ending attacker contract balance: ", endingAttackerContractBalance);
        console.log("Ending puppy raffle balance: ", endingPuppyRaffleBalance);

    }
```

Recommended Mitigation: To mitigate this vulnerability there are a few options:

1. move the storage update on players array prior to the external call

```
    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);
-       payable(msg.sender).sendValue(entranceFee);

+       payable(msg.sender).sendValue(entranceFee);
-       players[playerIndex] = address(0);
        emit RaffleRefunded(playerAddress);
    }
```

2. Use a openzeppelin ReentrancyGuard library to prevent reentrancy attacks.

```
+   function refund(uint256 playerIndex) public reentrancyGuard {
-   function refund(uint256 playerIndex) public {
```

3. Introduce a lock mechanism on the `PuppyRaffle::refund` function.

```
    function refund(uint256 playerIndex) public {

+       if (locked) {
+           revert("PuppyRaffle: Reentrancy detected");
+       }
+       locked = true;

        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);
    }
```

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

Description: The logic implemented to determine the winnerIndex used in the `PuppyRaffle::selectWinner` does not guarantee its randomness. The parameters used in the keccak256 execution such as `block.timestamp` and `block.difficulty` could be manipulated by miners, and since `msg.sender` is part of the hash, the caller can predict the outcome before submitting the transaction.

Impact: The weak randomness implementation allows malicious users to manipulate and predict the winner selection process. This enables them to: 1. Guarantee winning the prize pool by calculating the optimal time to call selectWinner() 2. Choose which puppy NFT they want by manipulating the winning index 3. Undermine the rarity system since winners can effectively pick any puppy they want

This breaks the core functionality and fairness of the raffle system.

Proof of Concept: Attack vectors: 1. Miners/validators can manipulate the block values (`timestamp`, `difficulty`) to influence the winner selection in their favor, since these values are used in the random number generation. 2. Since `msg.sender` is included in the winner calculation hash, users can create multiple addresses and simulate transactions to find one that would make them the winner before actually submitting the winning transaction. 3. The combination of predictable inputs allows malicious actors to effectively choose the winner by controlling transaction timing and sender address.

Recommended Mitigation: Consider using a more secure randomness source, 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 integer overflows.

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

Impact: In `PuppyRaffle::selectWinner`, `totalFees` are accumulated for the `feeAddress` to collect later in `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 first conclude a raffle of 4 players to collect some fees. 2. We then have 89 additional players enter a new raffle, and we conclude that raffle as well. 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 now not be able to withdraw, due to this 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 the fees, this is clearly not what the protocol is intended to do.

Proof Of Code

Place this into the `PuppyRaffleTest.t.sol` file.

```
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 recommended mitigations here.

1. Use a newer version of Solidity that does not allow integer overflows by default.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's `SafeMath` to prevent integer overflows.

2. Use a `uint256` instead of a `uint64` for `totalFees`.

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

3. Remove the balance check in `PuppyRaffle::withdrawFees`

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

We additionally want to bring your attention to another attack vector as a result of this line in a future finding.

Medium

[M-1] Looping thorough players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potencial denial of service (DoS) attack, incrementing gas costs for future entrants

Description: The `PuppyRaffle::enterRafflee` function loops through 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 players who enter 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.

```
// @audit DoS Attack
@>   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 gas cost for 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.

And attacker input make the `PuppyRaffle::entrants` array so big, taht no one ese enters, guarenteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas cost will be as such: - 1st 100 players: ~6503275 - 2st 100 players: ~18995515 This is more than 3 times more expensive for the same 100 players.

PoC

```
    function test_denialOfService() public {
        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(i);
        }
        uint256 gasStart = gasleft();
        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
            );

        // now for the 2nd 100 players
        address[] memory playersTwo = new address[](playersNum);
        for(uint256 i = 0; i < playersNum; i++) {
            playersTwo[i] = address(i + playersNum);
        }
        uint256 gasStartSecond = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(
            playersTwo);
        uint256 gasEndSecond = gasleft();
        uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
            gasprice;
        console.log("Gas cost of the 2nd 100 players: ", gasUsedSecond)
            ;

        assert(gasUsedFirst < gasUsedSecond);
    }
```

Recommended Mitigation:

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same perso nfrom entering multiple times, only the same wallet address.
2. Consider using a mappping to check for duplicates. This would be more gas efficient, and would not require a loop.

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

     function selectWinner() external {
+        raffleId = raffleId + 1;
         require(block.timestamp >= raffleStartTime + raffleDuration, "
            PuppyRaffle: Raffle not over");
```

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

[M-2] Balance validation in `PuppyRaffle::withdrawFees` enables denial of service through forced ETH transfers

Description: The `PuppyRaffle::withdrawFees` function contains a strict validation that forces the contract's ETH balance to be exactly equal to `totalFees`. Although the contract lacks explicit payable functions to receive ETH, attackers can still manipulate the contract's balance by using `selfdestruct()` to forcefully send ETH, which circumvents this balance check and causes the validation to fail.

```
     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");
     }
```

Impact: This would cause a DoS prevening the feeAddress from withdrawing the fees.

Proof of Concept: Consider the steps bellow: - PuppyRaffle has 1 ETH in it's balance, and 1 ETH totalFees. - Attacher sends 1 wei(or any amount) via a selfdestruct - feeAddress is no longer able to withdraw funds

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

```
     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] 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;
```

Informational

[I-1] Floating pragmas

Description: Using floating pragma (`^0.7.6`) is dangerous. The contract can be compiled with any 0.7.x version, which may have different behaviors and security fixes. This could lead to untested or vulnerable code in production. Lock the pragma to a specific version.

https://swcregistry.io/docs/SWC-103/

Recommended Mitigation: Lock up pragma versions.

```
-   pragma solidity ^0.7.6;
+   pragma solidity 0.7.6;
```

[I-2] No validation for zero length players

Description: The `PuppyRaffle::enterRaffle` function does not check if the `newPlayers` array is empty. This allows users to send ether and not receive a position in the raffle.

```
function enterRaffle(address[] memory newPlayers) public payable {
//@>     @audit No validation for zero length players
         require(msg.value == entranceFee * newPlayers.length, "
             PuppyRaffle: Must send enough to enter raffle");
         for (uint256 i = 0; i < newPlayers.length; i++) {
```

Impact: Users can send ethers and not receive a position in the raffle. Causing a loss of funds.

Proof of Concept: Include the test in the `PuppyRaffleTest.t.sol` file and run it with `forge test --match-test test_zeroLengthPlayers` to check the impact fisibility.

PoC

```
    function test_zeroLengthPlayers() public {
        address[] memory players = new address[](1);
        uint256 contractBalanceBefore = address(puppyRaffle).balance;
        puppyRaffle.enterRaffle{value: entranceFee}(players);

        uint256 contractBalanceAfter = address(puppyRaffle).balance;
        assert(contractBalanceAfter > contractBalanceBefore);
    }
```

Recommended Mitigation: Add a check to the `PuppyRaffle::enterRaffle` function to check if the `newPlayers` array is empty.

```
    function enterRaffle(address[] memory newPlayers) public payable {
+       require(newPlayers.length > 0, "PuppyRaffle: Players array
    cannot be empty");
        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]);
        }
```

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

Description: Assigning values to address state variables without checking for `address(0)`.

- Found in src/PuppyRaffle.sol: 8662:23:35
- Found in src/PuppyRaffle.sol: 3165:24:35
- Found in src/PuppyRaffle.sol: 9809:26:35

[I-4] Magic Numbers

Description: All number literals should be replaced with constants. This makes the code more readable and easier to maintain. Numbers without context are called "magic numbers".

Recommended Mitigation: Replace all magic numbers with constants.

```
+   uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
+   uint256 public constant FEE_PERCENTAGE = 20;
+   uint256 public constant TOTAL_PERCENTAGE = 100;
-   uint256 prizePool = (totalAmountCollected * 80) / 100;
-   uint256 fee = (totalAmountCollected * 20) / 100;
    uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE)
        / TOTAL_PERCENTAGE;
    uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) /
        TOTAL_PERCENTAGE;
```

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::legendaryUri` should be `constant`