Protocol Audit Report

Version 1.0

Cyfrin.io

October 9, 2024

PuppyRaffle Audit Report

CC

October 9, 2024

Prepared by: CasinoCompiler Lead Auditors: - CC

Table of Contents

  - [M-2] Denial of Service
  - [M-3] Unsafe cast of `PuppyRaffle::fee` loses fees
  - [M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` will block the start of a new contest

- Low
  - [L-1] `PuppyRaffle.sol::getActivePlayerIndex()` returns 0 if player is inactive leading to misintepretation.

- Gas
  - [G-1] Use of storage variables for persistant variables
  - [G-2] For loops reading from storage for each iteration in `PuppyRaffle.sol::enterRaffle()`

- Informational
  - [I-1] Use of floating pragma can lead to unexpected bahaviour
  - [I-2] Use an older version of solidity is not recommended
  - [I-3] Missing check for `address(0)` when assigning values to address state variables.
  - [I-4] Multiple users not possible.
  - [I-5] `PuppyRaffle.sol::selectWinner()` does not follow CEI.
  - [I-6] Naming convention
  - [I-7] Use of magic numbers

- Issues I did not find
  - [High] Malicious winner can forever halt the raffle

Protocol Summary

The smart contract creates a raffle system that starts the moment the contract is deployed and has a duration which is set during contract deployment. Players can enter themselves or multiple different wallets with a fixed entrance fee, which is also set during deployment; no duplicate wallet addresses are allowed. once the raffle duration limit has superseded, the winner can be called manually if there are atleast 4 players within the raffle, upon which the winner will receive 80% of the total fees the raffle collected as well as an NFT of random rarity. the remaining 20% of the total fees can be collected by the raffle owner to a collection address which can be altered.

Disclaimer

The CC team 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 by the team 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 to 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

As is, the current architecture of the protocol should not be implemented and be redesigned. By nature, all data on a blockchain is public information, thus, sensitive data should not be stored in raw form.

Issues found

Severity	Number of Issues Found
High	4
Medium	4
Low	1
Gas	2
Informational	7
Total	18

Findings

High

[H-1] Reentrency in `PuppyRaffle.sol:refund()` allowing for total contract drainage.

Description:

`PuppyRaffle.sol::refund()` does not follow the CEI function implementation structure, and as a result, a reentrency vector is introduced.

when `PuppyRaffle.sol::refund()` is called, it sends the refund via an external call to the user index before state is updated i.e. removing the player from the raffle.

Impact:

Total drainage of `PuppyRaffle.sol` contract.

Proof of Concept: A bad actor can create an attack contract with a `receive()/fallback()` function that will

1. Enter so it is in the raffle system and then

2. Repeatedly call `PuppyRaffle.sol::refund()` until the contract is drained.

Recommended Mitigation:

To prevent this, we should have the `PuppyRaffle.sol::refund()` function update the `PuppyRaffle.sol::players` array before making the external call. Additionally, the event emissions should also be moved up.

Modification to PuppyRaffle.sol::refund()

```
    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 for `PuppyRaffle.sol::selectWinner()`, the winner can be easily gamed.

Description:

The raffle system can be gamed as using `msg.sender`, `block.timestamp` and `block.difficulty` as a means to generate a random number is flawed as all 3 variables can easily be manipulated by a node validator, a player or a player working with a validator.

Impact:

Both the winner and the NFT rarity are determined using this pseudo-randomness, thus the system can be gamed so that a bad actor will only enter if they can guarantee: a. their winnings b. a rare NFT

Proof of Concept:

1. A bad actor will calulate the winner with manipulated `msg.sender`, `block.timestamp` and `block.difficulty`.
2. They will enter the raffle when they believe is optimal timing
3. call `PuppyRaffle.sol::selectWinner()` and win the raffle

Recommended Mitigation:

Consider using Chainlink VRF, a proveable random seed generator.

[H-3] Frontrunning `PuppyRaffle.sol::selectWinner()` in the case of an unfavourable outcome

Description:

As a result of the pseudo-randomness, `PuppyRaffle.sol::selectWinner()` can be frontrun for multiple reasons:

1. raffle winner was not the bad actor therefore they refund their entry
2. raffle winner did not win a rare NFT therefore they refund their entry

Impact:

Integrity of the raffle is compromised. Additionally, the raffle may never actually conclude if every entree were running a script to check for unfavourable outcomes.

Proof of Concept:

1. Attacker joins raffle
2. Attacker creates script listening for `selectWinner()` to be called and calculate the winner.
3. If the attacker != winner, he can frontrun the txn in the mempool by submitting `refund()` with higher gas, thus ensuring his txn is ordered before `selectWinner()` execution.

Recommended Mitigation:

The root cause of the issue is the randomness, the mwthodology should be revised.

[H-4] Arithmatic overflow of `PuppyRaffle::totalFees` loses fees.

Description:

Prior to solidity `0.8.0`, arithmatic overflow could occur if the value were to surpass the maximum value for the casted type.

Example

```
1     uint64 myVar = type(uint64).max; //18446744073709551615
2     myVar += 1; // Expected value 18446744073709551616 | Actual value:
          0
```

Impact:

The `totalFees` are accumulated using this expression [1] and totalFees itself is of type uint64; uint64 is considered small and could easily overflow, thus the incorrect fees would be collected by the `feeAddress`.

Proof of Concept:

1. Assume the `entranceFee` is `30 ether`.
2. If 4 players were to enter the raffle, using [1], the `totalFees` would be calculated as:

   $\text{totalFees}_{\text{Expected}} = ((30e18 * 4) * 20)/100$

   $\text{totalFees}_{\text{Expected}} = 24e18$
3. However, the actual totalFees will be a different value due to the overflow.

   $\text{totalFess}_{\text{Actual}} = 5553255926290448384$

   $\text{totalFess}_{\text{Actual}} \approxeq 55.53e17$
4. As a result, you will not be able to withdraw the totalFees as you will not pass the first check in `PuppyRaffle::withdrawFees()`.

Recommended Mitigation:

1. Use a newer version on solidity which has SafeMath built in by default. If solc `<0.8.0` is still being used, Safemath can be imported and used to safeguard from overflow.
2. Use a larger type of uint i.e. `uint256`.

Medium

[M-1] The fees accrued can be stuck in the contract indefinitely due to a strict equality check in `PuppyRaffle.sol::withdrawFunds()`

Description: The `PuppyRaffle::withdrawFees` function checks the `totalFees` equals the ETH balance of the contract (`address(this).balance`). Since this contract doesn't have a `payable` fallback or `receive` function, you'd think this wouldn't be possible, but a user could `selfdesctruct` a contract with ETH in it and force funds to the `PuppyRaffle` contract, breaking this check.

```
1      function withdrawFees() external {
2  @>      require(address(this).balance == uint256(totalFees), "
       PuppyRaffle: There are currently players active!");
3          uint256 feesToWithdraw = totalFees;
```

```
4           totalFees = 0;
5           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6           require(success, "PuppyRaffle: Failed to withdraw fees");
7       }
```

Impact: This would prevent the `feeAddress` from withdrawing fees. A malicious user could see a `withdrawFee` transaction in the mempool, front-run it, and block the withdrawal by sending fees.

Proof of Concept:

1. `PuppyRaffle` has 800 wei in it's balance, and 800 totalFees.
2. Malicious user sends 1 wei via a `selfdestruct`
3. `feeAddress` is no longer able to withdraw funds

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

```
1       function withdrawFees() external {
2   -       require(address(this).balance == uint256(totalFees), "
        PuppyRaffle: There are currently players active!");
3           uint256 feesToWithdraw = totalFees;
4           totalFees = 0;
5           (bool success,) = feeAddress.call{value: feesToWithdraw}("");
6           require(success, "PuppyRaffle: Failed to withdraw fees");
7       }
```

[M-2] Denial of Service

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 that 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.

Note to students: This next line would likely be it's own finding itself. However, we haven't taught you about MEV yet, so we are going to ignore it. Additionally, this increased gas cost creates front-running opportunities where malicious users can front-run another raffle entrant's transaction, increasing its costs, so their enter transaction fails.

Impact: The impact is two-fold.

1. The gas costs for raffle entrants will greatly increase as more players enter the raffle.
2. Front-running opportunities are created for malicious users to increase the gas costs of other users, so their transaction fails.

Proof of Concept:

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: 6252039 - 2nd 100 players: 18067741

This is more than 3x as expensive for the second set of 100 players!

This is due to the for loop in the `PuppyRaffle::enterRaffle` function.

```
1          // Check for duplicates
2  @>      for (uint256 i = 0; i < players.length - 1; i++) {
3              for (uint256 j = i + 1; j < players.length; j++) {
4                  require(players[i] != players[j], "PuppyRaffle:
                       Duplicate player");
5              }
6          }
```

Proof Of Code

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

```
1  function testReadDuplicateGasCosts() public {
2          vm.txGasPrice(1);
3
4          // We will enter 5 players into the raffle
5          uint256 playersNum = 100;
6          address[] memory players = new address[](playersNum);
7          for (uint256 i = 0; i < playersNum; i++) {
8              players[i] = address(i);
9          }
10         // And see how much gas it cost to enter
11         uint256 gasStart = gasleft();
12         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               players);
13         uint256 gasEnd = gasleft();
14         uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
15         console.log("Gas cost of the 1st 100 players:", gasUsedFirst);
16
17         // We will enter 5 more players into the raffle
18         for (uint256 i = 0; i < playersNum; i++) {
19             players[i] = address(i + playersNum);
20         }
21         // And see how much more expensive it is
22         gasStart = gasleft();
23         puppyRaffle.enterRaffle{value: entranceFee * playersNum}(
               players);
24         gasEnd = gasleft();
25         uint256 gasUsedSecond = (gasStart - gasEnd) * tx.gasprice;
26         console.log("Gas cost of the 2nd 100 players:", gasUsedSecond);
27
28         assert(gasUsedFirst < gasUsedSecond);
29         // Logs:
```

```
30          //     Gas cost of the 1st 100 players: 6252039
31          //     Gas cost of the 2nd 100 players: 18067741
32  }
```

Recommended Mitigation: There are a few recommended mitigations.

1. Consider allowing duplicates. Users can make new wallet addresses anyways, so a duplicate check doesn't prevent the same person from entering multiple times, only the same wallet address.
2. 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; 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");
```

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

[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.

```
1      function selectWinner() external {
2          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
3          require(players.length > 0, "PuppyRaffle: No players in raffle"
               );
4
5          uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.
               sender, block.timestamp, block.difficulty))) % players.
               length;
6          address winner = players[winnerIndex];
7          uint256 fee = totalFees / 10;
8          uint256 winnings = address(this).balance - fee;
9  @>      totalFees = totalFees + uint64(fee);
10         players = new address[](0);
11         emit RaffleWinner(winner, winnings);
12     }
```

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.

```
1  -   uint64 public totalFees = 0;
2  +   uint256 public totalFees = 0;
3  .
4  .
5  .
6      function selectWinner() external {
7          require(block.timestamp >= raffleStartTime + raffleDuration, "
               PuppyRaffle: Raffle not over");
8          require(players.length >= 4, "PuppyRaffle: Need at least 4
               players");
9          uint256 winnerIndex =
10             uint256(keccak256(abi.encodePacked(msg.sender, block.
                   timestamp, block.difficulty))) % players.length;
11         address winner = players[winnerIndex];
12         uint256 totalAmountCollected = players.length * entranceFee;
13         uint256 prizePool = (totalAmountCollected * 80) / 100;
14         uint256 fee = (totalAmountCollected * 20) / 100;
15 -       totalFees = totalFees + uint64(fee);
16 +       totalFees = totalFees + fee;
```

[M-4] Smart Contract wallet raffle winners without a `receive` or a `fallback` 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::selectWinner` 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 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 so winners can pull their funds out themselves, putting the owness on the winner to claim their prize. (Recommended)

Low

[L-1] `PuppyRaffle.sol::getActivePlayerIndex()` returns 0 if player is inactive leading to misintepretation.

Description:

`PuppyRaffle.sol::getActivePlayerIndex()` returns 0 if an address is not an active player.

Impact:

This can be misinterpreted as someone who isn't active may think they are index 0 or the first player who entered the raffle think they did not enter the raffle.

In the case of the latter, the player may try to enter again but `PuppyRaffle.sol::enterRaffle()` will revert as he is a duplicate player, therefore, the player will waste gas.

Recommended Mitigation:

`PuppyRaffle.sol::_isActivePlayer()` is an unused internal function with the recommended mitigation but implemented incorrectly. > **NOTE:** Modifying original function is recommended - leaving original function as internal makes it redundent code which will also increase gas cost for contract deployment.

modify PuppyRaffle.sol::_isActivePlayer() to be a public view function

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

Gas

[G-1] Use of storage variables for persistant variables

Description: The following varaibles are all marked as storage when they are unchanged after first implementation: a. `PuppyRaffle.sol::raffleDuration` b. `PuppyRaffle.sol::commonImageUri` c. `PuppyRaffle.sol::rareImageUri` d. `PuppyRaffle.sol::legendaryImageUri`

More gas is required when executing functions that read these variables.

Recommended Mitigation:

Store a. as an immutable variable and c->d as constant.

[G-2] For loops reading from storage for each iteration in `PuppyRaffle.sol::enterRaffle()`

Description:

The duplicate player checker reads `PuppyRaffle.sol::players[]` for each iteration of the loop. Storage reads are more gas expensive and for-loops are (O)^n computationally.

Recommended Mitigation:

Cache the players length in memory

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

> **NOTE:** This is under the assumption [H-DoS] mitigations were not implemented or a similar solution with for loops were still used - in which case, consider caching storage values if required.

Informational

[I-1] Use of floating pragma can lead to unexpected bahaviour

Description: Using a wide range of solidity can lead to unexpected behaviour.

Recommended Mitigation: Consider using a specified version of solidity and avoid ^

[I-2] Use an older version of solidity is not recommended

Description: Industry practice to use latest version of solidity or >=0.8.18.

Recommended Mitigation: Consider using the latest version of solidity.

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

Description:

Missing `address(0)` check for setting `PuppyRaffle.sol::feeAddress` in first implementation through constructor and also when `PuppyRaffle.sol::changeFeeAddress` is called. In the first instance, if no feeAddress is provided during contract creation, address defualts to `adderess(0)`.

Impact: Low

Recommended Mitigation:

1. Set a check in the constructor as such:

```
    constructor(uint256 _entranceFee, address _feeAddress, uint256 _raffleDuration) ERC721("Puppy Raffle", "PR") {
+       if (_feeAddress == address(0)){
+           revert();
        }
        ... // Rest of code
```

2. Set a check in PuppyRaffle.sol::changeFeeAddress

```
    function changeFeeAddress(address newFeeAddress) external onlyOwner {
+       if (newFeeAddress == address(0)){
+           revert();
        }

        feeAddress = newFeeAddress;
        emit FeeAddressChanged(newFeeAddress);
    }
```

[I-4] Multiple users not possible.

Description:

As per the project documentation:

> "A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends."

Multiple users is not actually possible due to the duplicate players search.

Recommended Mitigation:

If the project intended to allow multiple entries, the duplicate players check in `PuppyRaffle.sol::enterRaffle()` should be removed.

Removing this has several benefits:

1. Removes the duplicate players check which was shown in [H-Dos] to be a High security vulnerability.
2. Intended funtionality implemented.
3. Not allowing duplicate addresses doesn't necessarily remove duplicate players; players can enter through other owned address/es.

[I-5] `PuppyRaffle.sol::selectWinner()` does not follow CEI.

Description:

Best practice is to always follow CEI, even if reentrency doesn't seem plausible.

Recommended Mitigation:

Modification to PuppyRaffle.sol::selectWinner()

```
1     function selectWinner() external {
2         require(block.timestamp >= raffleStartTime + raffleDuration, "
              PuppyRaffle: Raffle not over");
3         require(players.length >= 4, "PuppyRaffle: Need at least 4
              players");
4         uint256 winnerIndex =
5             uint256(keccak256(abi.encodePacked(msg.sender, block.
                  timestamp, block.difficulty))) % players.length;
6         address winner = players[winnerIndex];
7         uint256 totalAmountCollected = players.length * entranceFee;
8         uint256 prizePool = (totalAmountCollected * 80) / 100;
9         uint256 fee = (totalAmountCollected * 20) / 100;
10        totalFees = totalFees + uint64(fee);
11
12        uint256 tokenId = totalSupply();
13
14        // We use a different RNG calculate from the winnerIndex to
              determine rarity
15        uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
              block.difficulty))) % 100;
```

```
16          if (rarity <= COMMON_RARITY) {
17              tokenIdToRarity[tokenId] = COMMON_RARITY;
18          } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
19              tokenIdToRarity[tokenId] = RARE_RARITY;
20          } else {
21              tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
22          }
23
24          delete players;
25          raffleStartTime = block.timestamp;
26          previousWinner = winner;
27  +       _safeMint(winner, tokenId);
28          (bool success,) = winner.call{value: prizePool}("");
29          require(success, "PuppyRaffle: Failed to send prize pool to
                winner");
30  -       _safeMint(winner, tokenId);
31      }
```

[I-6] Naming convention

Description:

Smart contract conventions not followed for variables.

Recommendation:

Storage and immutable variables prefixed with s_ & i_, respectfully.

Constant variables are fully capitalised.

[I-7] Use of magic numbers

Description:

Recommended not to use magic numbers in code blocks and clearly state variables beforehand. This is better for readability as well as modularity.

Recommended Mitigation:

In `PuppyRaffle.sol::selectWinner()`, clearly define magic numbers in lines 132 133 139

Issues I did not find

High Malicious winner can forever halt the raffle

Description: Once the winner is chosen, the `selectWinner` function sends the prize to the the corresponding address with an external call to the winner account.

```
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
```

If the `winner` account were a smart contract that did not implement a payable `fallback` or `receive` function, or these functions were included but reverted, the external call above would fail, and execution of the `selectWinner` function would halt. Therefore, the prize would never be distributed and the raffle would never be able to start a new round.

There's another attack vector that can be used to halt the raffle, leveraging the fact that the `selectWinner` function mints an NFT to the winner using the `_safeMint` function. This function, inherited from the `ERC721` contract, attempts to call the `onERC721Received` hook on the receiver if it is a smart contract. Reverting when the contract does not implement such function.

Therefore, an attacker can register a smart contract in the raffle that does not implement the `onERC721Received` hook expected. This will prevent minting the NFT and will revert the call to `selectWinner`.

Impact: In either case, because it'd be impossible to distribute the prize and start a new round, the raffle would be halted forever.

Proof of Concept:

Proof Of Code

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

```
function testSelectWinnerDoS() public {
    vm.warp(block.timestamp + duration + 1);
    vm.roll(block.number + 1);

    address[] memory players = new address[](4);
    players[0] = address(new AttackerContract());
    players[1] = address(new AttackerContract());
    players[2] = address(new AttackerContract());
    players[3] = address(new AttackerContract());
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    vm.expectRevert();
    puppyRaffle.selectWinner();
}
```

For example, the `AttackerContract` can be this:

```
contract AttackerContract {
    // Implements a `receive` function that always reverts
    receive() external payable {
        revert();
    }
}
```

Or this:

```
contract AttackerContract {
    // Implements a `receive` function to receive prize, but does not
       implement `onERC721Received` hook to receive the NFT.
    receive() external payable {}
}
```

Recommended Mitigation: Favor pull-payments over push-payments. This means modifying the `selectWinner` function so that the winner account has to claim the prize by calling a function, instead of having the contract automatically send the funds during execution of `selectWinner`.