23.03.2024

Introduction

A time-boxed security review of the first-flight **Puppy-Raffle** protocol was done by **hashov**, focused on the security aspects of the application's implementation.

Disclaimer

A smart contract security review cannot ensure the total elimination of vulnerabilities. This process is limited by time, resources, and expertise, as I strive to identify as many vulnerabilities as I can for the given timeframe. I cannot promise 100% absolute security post-review or guarantee the discovery of any issues in your smart contracts. It is highly advised to conduct additional security reviews, implement bug bounty programs, and engage in on-chain monitoring for enhanced protection.

About hashov

is a first-flight dedicated individual who specializes in auditing blockchain technology independently. By identifying multiple security flaws in different protocols, he actively works towards enhancing the security of the blockchain ecosystem through thorough research and reviews. Feel free to reach out on Twitter @hashovweb3

About Puppy-Raffle

Link to the project can be found here.

Is a protocol which 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.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

review commit hash **- e30d199697bbc822b646d76533b66b7d529b8ef5**

Scope

The following smart contracts were in scope of the audit:

- `PuppyRaffle.sol`

The following number of issues were found, categorized by their severity:

- Critical 2 issues
- High: 2 issues
- Medium: 4 issues
- Low: 1 issues
- Informational: 3 issues

Findings Summary

ID	Title	Severity
[C-01]	Possible reentrancy inside `refund` function	Critical
[C-02]	Possible reentrancy inside `selectWinner` function	Critical
[H-01]	Denial of service inside `enterRiffle` function on iteration with nested loops over array	High
[H-02]	Mishandling of ETH inside `withdrawFees`	High
[M-01]	Weak randomness inside `selectWinner` function, cause of `block.timestamp` usage	Medium
[M-02]	Possible overflow on `fees` variable	Medium
[M-03]	Unsafe casting from `uint256` to `uint64`	Medium
[M-04]	Weak randomness exploit. Using `block.timestamp`	Medium

ID	Title	Severity
[L-01]	The `getActivePlayerIndex` function returns 0 if player is not active	Low
[I-01]	Immutable variables naming convention	Info
[I-02]	Prefer usage of `msg.sender` instead `address` parameter inside constructor	Info
[I-03]	Consider usage of Solidity version over `0.8.*` and floating pragma removal	Info

Detailed Findings

[C-01] Possible reentrancy inside refund() function

Severity

Impact: High, as it results in 100% loss for funds of targeted players inside the protocol

Likelihood: High, as any user of `Puppi Raffle` with malicious intention can interact violently with the contract.

Description

Function `refund()` uses `payable(msg.sender).sendValue` coming from `openZeppelin` library, which is a great way to surpass the Solidity's `transfer()` and `send()` methods from running out of gas, but still this line is open to reentrancy. If the calling contract does not have a way to handle native currency, `fallback()` will be called therefore giving him the opportunity to reenter before execution of `refund()` finishes. Of course with the usage of arbitrary `playerIndex` and sequential calls to `refund()` attacker can withdraw players funds out of the contract.

Recommendations

1. Do all state changes before doing the call

2. Add reentrancy guard modifier such as:

```
    bool internal locked;


    modifier noReentrant() {
        require(!locked, "No re-entrancy");
        locked = true;
        _;
        locked = false;
    }
```

[C-02] Possible reentrancy inside selectWinner() function

Severity

Impact: High, as it results in 100% loss of `prizePool` funds

Likelihood: High, as any user of `Puppi Raffle` protocol with malicious intention can interact violently with the contract.

Description

Function `selectWinner()` uses `winner.call{value: prizePool}("")` before doing state changes making it open to reentrancy. If the calling contract does not have a way to handle native currency, `fallback()` will be called therefore giving him the opportunity to reenter and do damage.

Recommendations

Add reentrancy guard modifier such as:

```
    bool internal locked;

    modifier noReentrant() {
        require(!locked, "No re-entrancy");
        locked = true;
        _;
        locked = false;
    }
```

or do all state changes before doing the call.

[H-01] Denial of service inside enterRaffle() function on iteration with nested loops over array

Severity

Impact: High, as the contract usage will be totally inefficient.

Likelihood: High, as anyone with malicious intend can call `enterRaffle()` with huge non-duplicating array, making the protocol inefficient.

Description

The contract stops being usable if malicious user calls `enterRaffle()` with huge non-duplicating array and `newPlayers` array would be really heavy on the following piece of code on line 81:

```
// Check for duplicates
        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");
            }
        }
```

making it big enough and gas expensive therefore making transactions fail and making it unusable.

Recommendations

1. First do the duplicate player entry check and then add people to the players array then add test case for that, this way you will minimalize the risk of someone maliciously filling up your array, but in future at some point the array will potentially get large enough to cause problems.
2. Usage of: `mapping(address => uint256) public addressToRaffleId;` in conjuction with `uint256 public raffleId = 0;` is encouraged. This solution will give instant check if someone who is trying to enter is a duplicate and we will solve gas efficiency problems.

[H-02] Mishandling of ETH inside withdrawFees()

Severity

Impact: High, as the contract usage will be totally inefficient.

Likelihood: High, as anyone with malicious intend can call `enterRaffle` with huge non-duplicating array, making the protocol inefficient.

Description

Since the contract does not have implemented `fallback()` or `receive()` methods a potential attacker can force eth inside of the contract with `selfdestruct()` function and the following check inside `withdrawFees()` would always fail:

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

this will lead to failing of the given assertion always when someone tries to withdraw, therefore nobody will be able to withdraw native token from the contract.

Recommendations

Don't rely on `address(this).balance` instead of, make usage of storage variable such as: `uint256 public balance;`

[M-01] Weak randomness inside selectWinner() function, cause of block.timestamp usage

Severity

Impact: Low, as it will not have any impact on the contract itself but it will mislead client of the contract.

Likelihood: High, as anyone has the chance to not be included in the raffle.

Description

Using `block.timestamp` inside of `selectWinner()` on line 126:

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

for source of randomness can always be manipulated by attacker, cause the number can be guessed by using the same formula.

Recommendations

As a solution usage of Chainlink VRF (Verifiable Random Function) might be preferable.

[M-02] Possible overflow on fees variable

Severity

Impact: High, owner of contract will receive 0 `fees` if variable overflows.

Likelihood: High, as the maximum number of `uint64` is not high enough to hold, large volume of fees.

Description

Overflow on `totalFees` storage variable is possible, cause the number of fees depends on how many people are in the raffle, meaning it can surpass the max value of `uint64`, and since the solidity version is less than version `^0.8` it means that a check for surpass won't be present and owner of contract will receive 0 fees if it overflows.

Recommendations

1. Newer version of solidity
2. `Openzeppin`'s `SafeMath` library
3. `uint256` usage instead of `uint64`

[M-03] Unsafe casting from uint256 to uint64

Severity

Impact: High, owner of contract will receive 0 `fees` if variable overflows.

Likelihood: High, as the maximum number of `uint64` is not high enough to hold, large volume of fees.

Description

Inside of `electWinner()` line 132:

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

as shown `totalFees` fee is getting casted and will lose value due to that.

Recommendations

Remove the casting.

[M-04] Weak randomness inside selectWinner() function, cause of block.difficulty usage

Severity

Impact: Low, as it will not have any impact on the contract itself but it will mislead client of the contract.

Likelihood: High, as anyone has the chance to not be included in the raffle.

Description

Using `block.difficulty` for the rarity calculation inside of `selectWinner()` on line: 137:

```
 uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender,
block.difficulty))) % 100;
```

for source of randomness always can be manipulated, cause the number can be guessed by using the same formula.

Recommendations

As a solution usage of Chainlink VRF (Verifiable Random Function) might be preferable.

[L-01] The getActivePlayerIndex() function returns 0 if player is not active

Severity

Impact: Low, as it will not have any impact on the contract itself but it will mislead client of the contract.

Likelihood: High, as anyone has the chance to not be included in the raffle.

Description

the `getActivePlayerIndex()` function returns 0 if player is not active, what if player is at index 0? He might think he is not active at all.

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

this will lead to failing of the given assertion always when someone tries to withdraw, therefore nobody will be able to withdraw native token from the contract.

Recommendations

Consider using custom exception for example:

```
function getActivePlayerIndex(address player) external view returns (uint256) {
       for (uint256 i = 0; i < players.length; i++) {
           if (players[i] == player) {
               return i;
           }
           throw PlayerNotActive();
       }
   }
```

[I-01] Immutable variables naming convention

Severity

Informational

Description

Following best practices immutable variables such as:

```
    uint256 public immutable entranceFee;
```

should start with `i_` example: `i_entranceFee`

Recommendations

Applying best practices.

[I-02] Prefer usage of msg.sender instead address parameter inside constructor

Severity

Informational

Description

Instead of passing the address in the constructor since the one who will deploy the contract (owner) is the one who will get the winnings better to offload the constructor from the parameter to go with:

```
feeAddress = msg.sender;
```

Recommendations

Applying best practices.

[I-03] Consider usage of Solidity version over 0.8.* and floating pragma removal

Severity

Informational

Description

Increase version to solidity `0.8.*` to throw an exception if overflow/underflow occurs, also using floating pragma is a bad practice, since we want to know exactly with what version of solidity we are working.

Recommendations

Applying best practices.