Game.sol Audit Report

Version 0.1

Manar Maher

October 30, 2024

Puppy Raffle Auditing Report

Manarmaher1

Oct 30, 2024

Puppy Raffle Audit Report

Prepared by: Manar Maher
Lead Auditor: Manar Maher

Table of contents

See table

- Findings
  - High
  - Medium
  - Informational / Non-Critical
  - Gas (Optional)

Disclaimer

Manarmaher1 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

Audit Details

Commit hash:

```
```

Scope:

```
./src/
|_ Game.sol
```

Protocol Summary

Game.sol. The main smart contract, Game, is a game where users attempt to guess a value that produces a target hash. It is intended to have the following behavior:

1. The variable target is the goal hash that the user has to "guess."

2. A user provides their guess by issuing an attempt. The user's guess is not used directly, rather it is fed into a computation (in this case it simply adds guess to another value) and then the resulting hash is compared against target.

3. If the guesses match, then the user wins! When users make attempts, they have to pay cost for the right to guess. The winner is provided with all of the accrued funds from previous attempts to guess target.

4. After the game is won, the owner can initiate a new game by calling setTarget. While a target is not set, users cannot make any more attempts to guess the target

Roles

Owner : Deployer of the protocol, is the who intitiates new games and sets the target hash or updates it.
Player : the (guesser), is a user who interacts with the smart contract by attempting to guess the target hash.

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	1
Low	1
Info	3
Total	0

Findings

High

[H-1]] Reentrancy attack Game.sol::attempt allows players to drain the contract's balance

Description: The `Game.sol::attempt` function does not follow CEI guildine or the checks, effects, interaction. Which would cause a user to keep keep calling the function before the the state of the contract is updated to false, and as a result, enables players to drain the contract balance.

In the `Game.sol::attempt` function, we first make an external call to the `msg.sender` address, and

```
function attempt(uint guess) external payable {
require(started);
require(msg.value > cost);
value = value.add(guess);
bytes32 result = sha256(abi.encode(value));
if(result == target) {
 payable(msg.sender).send(address(this).balance);
started = false; }
```

only after making that external call, we update the `contract started state to false to imply ending this cuurent game round.`

Impact: A player who has accessed the game and managed to make the correct guess could have a `fallback`/`receive` function that calls the `game.sol::attempt` function again and withdrawing more from the balance of the contract. They should only be allowed access to the funds which is from the previous attempted guesses plus their own, but in case the contract holds additional funds that is not intended to be given to the winner, the players could manipulate this to drain the whole balance.

Recommendation: follow the CEI guidleines. the state or the game is updated before making any external calls, and as security best practices consider adding a reentrancy guard.

[H-2] Missing access control on the `setTarget` function.

1. **Description:** Missing access control on the `setTarget` function.
   The way the game is designed, the owner should be the only person allowed to initiate a new game by calling the `setTarget` function. But due to the lack of access control, anyone can call the function and set a new Target, and initiate a new game.

```
function setTarget(bytes32 t) external { target = t; started = true; }
```

Impact : unauthorized access is giving to malicious users who shouldn't be allowed to, making them capable of disturping the game, manipulating the value of the target or initiating a new game. Only the contract owner should be accessed this privilege.

Recommended Mitigation: it’s recommended to add access control checks to the `setarget` function to restrict the access to only the owner and adding open zeppelin's only owner modifiers to the contract.

[M-1] Integer overflow in Game.sol::attempt function

1. **Description:** while openzeppelin's safe math is regularly used along the contract. The way the contract is designed is by each new player accessing the game, and making a new guess, it's added to the old value. So new value is an accumulation of all previous guess values + new guess value. And there's really no range to what that guess value should range within, or any input validation.

```
function attempt(uint guess) external payable {
require(started);
require(msg.value > cost);
 value = value.add(guess);
bytes32 result = sha256(abi.encode(value));
if(result == target) {
 payable(msg.sender).send(address(this).balance);
 started = false;
}
```

Impact : So in a scenario where a lot of users have entered the game and the current value is already at a high number that's very close to the maximum value, and the a new users also makes a new guess that's a high number, now our variable has exceeded its maximum value, and with use of safemath it will revert. However even in case of normal users who has no malicious intents, that would disturp the functionality of he contract, probably even causing a denial of service. A malicious user could also take advantage of this.

Recommended Mitigation: it’s recommended to add more input validation to the guess value, making sure that the guess value stays within a reasonable range that doesn’t expose the contract to intentional or unitentioal over/underflow, that might disturb the functionality of the game.

[L-1]Input validation on the ' msg.Value' value could be insuffiecient.

1. **Description:** in the contract, with each time a user attempt to participate and make a new attempt, we have a require of them paying a fee or entering a value that is bigger than the required cost

```
require(msg.value > cost);
```

However instead of requiring the user to enter the value that's equivalent of the required cost as in : require(msg.value= cost); or even require(msg.value>= cost);
There'e really no maximum value decalred to the fee the user could pay, so it's now left for the users to guess the cost since the specific value itself is not declared and it's left for the players to guess. However putting more input validation to the maximum value of msg.value would be better and help the contract from any potential disturbing of the game or cause of any denial of service or just unnecessary high fee that might discourage new players from entering.
Recommended Mitigation: add more input validation to the msg.value.

[I-1] Floating pragmas :

Game.sol should use a specific strict version of solidity. Locking the version ensures that contracts are not deployed with a different version of solidity than they were tested with. An incorrect version could lead to uninteded results.

[I-2] No Event Emission for Critical functions.
There are no emit events within the contract, adding event emissions for important actions like declaring a winner, ending a current game or resetting the target would add more clarity for the players

[I-3] consider using Tranfer istean of send in the `payable(msg.sender).send(address(this).balance)`, for better security and functionality for the contract.