Lottery Audit Report

0xOwain

August 2025

Contents

Lottery Audit Report

Version 1.0

0xOwain

August 25, 2025

Protocol Summary

This is a simple Ethereum lottery smart contract, where a user can enter a lottery by sending a specified amount of Ether to the contract. Once enough participants have entered, the contract owner can choose a random winner who will receive the prize pool of Ether.

Disclaimer

0Owain 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 Likelihood	High	Medium	Low
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

Scope

Repository: Solidity-Lottery-Contract

Commit Hash 98f354b41af55444a0912c4a828ae352554c47c3

Contracts in Scope: - `Lottery.sol`

cloc Summary Language files blank comment code Solidity 1 7 6 27

Out of Scope:
- N/A (only one Solidity contract present in repo)

Severity Criteria

High - Direct loss of funds or permanent lock of assets.
- Anyone can exploit (not just privileged roles).
- Breaks core protocol functionality.

Medium
- Causes significant disruption (DoS, griefing, governance failure).
- Exploitable under some conditions or requires privileged roles.
- Financial loss is possible but limited.

Low
- Minor issues: inefficiencies, gas waste, unclear logic, small inconsistencies.
- Doesn't threaten core security or funds.

Informational / Non-Critical
- Code style, readability, missing comments.
- Best practices (naming conventions, event emissions, input validation improvements).
- No security impact.

Summary of Findings

Severity	Number of issues found
High	2
Medium	4
Low	2
Informational	2
Gas Optimisations	1
Total	**11**

Tools Used

Manual Review: Line-by-line code analysis of `Lottery.sol`

Testing: Custom JavaScript tests with Web3

Static Analysis: Slither 0.4.17

- Reported weak PRNG, CEI violation, unbounded array growth → all covered in manual findings
- Flagged outdated compiler version → added as Informational finding

High

[H-1] Funds locked if manager is inactive

Description: The contract enforces access control on `pickWinner()` using the restricted modifier:

```
modifier restricted() {
    require(msg.sender == manager);
    _;
}
```

As a result, only the manager can call `pickWinner()` and release the contract's balance. If the manager: - Loses access to their wallet, - Stops maintaining the contract, - Or chooses not to act,

Then all funds remain permanently locked in the contract.

Impact: - High severity: Players' funds become unrecoverable - No way to resolve the game - Creates a centralised trust dependency on the manager's availability

This breaks decentralisation assumptions and poses serious risk in real-world deployments.

Proof of Concept: 1. Deploy the Lottery contract

2. Have multiple users enter (sending ETH to the contract)

```
await lottery.methods.enter().send({ from: accounts[1], value: web3.utils.toWei("1", "ether'
await lottery.methods.enter().send({ from: accounts[2], value: web3.utils.toWei("1", "ether'
```

3. Do not call `pickWinner()`

Result:

```
// Manager must call this
function pickWinner() public restricted { ... }
```

If the manager never calls this, the contract balance is locked indefinitely, and there is no exit path for users.

Recommended Mitigation: Allow players to withdraw their entry if the lottery is unresolved after as certain period of time:

```
uint public lastEntryTime;

function enter() public payable {
    require(msg.value > 0.01 ether);
    players.push(msg.sender);
    lastEntryTime = now;
}

function pickWinner() public {
    require(msg.sender == manager || now > lastEntryTime + 1 days, "Not authorized yet");
    ...
}
```

[H-2] Weak randomness

Description: The contract's `random()` function generates a "random" number using predictable inputs:

```
function random() private view returns (uint) {
    return uint(keccak256(block.difficulty, now, players));
}
```

This uses: - `block.difficulty` → often stable or guessable - `now` (i.e. `block.timestamp`) → can be influenced slightly by miners (±15 seconds) - `players` → publicly visible and user-controlled

Because all of these inputs are either predictable or manipulable, the result of the keccak256 hash is not truly random. An attacker (especially the manager or a miner) could predict or manipulate the outcome.

Impact: - The manager, who controls when `pickWinner()` is called, can repeatedly test different block timestamps to force a favourable outcome. - A miner could influence `block.timestamp` to bias the result. - A user could watch `players[]` and enter last, repeatedly, hoping to skew the hash outcome in their favour.

This breaks the fairness assumption of a lottery. In real-world use, this could be exploited to drain funds over time with near-zero risk.

Proof of Concept: The same inputs lead to the same winner being selected. To demonstrate:

```
// 1. Add 3 known players to the contract
await lottery.methods.enter().send({ from: accounts[0], value: web3.utils.toWei('1', 'ether'
await lottery.methods.enter().send({ from: accounts[1], value: web3.utils.toWei('1', 'ether'
await lottery.methods.enter().send({ from: accounts[2], value: web3.utils.toWei('1', 'ether'

// 2. Freeze block time to keep input values stable
await web3.currentProvider.send({ jsonrpc: "2.0", method: "evm_increaseTime", params: [0], i
await web3.currentProvider.send({ jsonrpc: "2.0", method: "evm_mine", id: 1 });

// 3. Call random() multiple times and compare results
const index1 = await lottery.methods.random().call();
const index2 = await lottery.methods.random().call();

assert.strictEqual(index1, index2); // Same inputs = same winner
```

Recommended Mitigation: Use a commit-reveal scheme or external randomness oracle (e.g., Chainlink VRF).

Medium

[M-1] State update after external call

Description: The `pickWinner()` function transfers the contract balance to the selected winner before resetting the `players` array:

```
players[index].transfer(this.balance);
players = new address ;
```

This violates the Checks-Effects-Interactions pattern and could introduce reentrancy risk if `players[index]` is a contract with a fallback function.

Impact: If the selected winner is a contract, it may execute arbitrary fallback logic before the contract state has been updated — including checking or modifying global variables.

While .transfer in Solidity 0.4.x limits gas to 2,300 (which mitigates full reentrancy), this design still opens the door for: - Future breakage (e.g., if `.call.value()` is used) - Unexpected side effects - Lower auditability and upgrade safety

Proof of Concept: 1. Deploy a malicious contract with a payable fallback:

```
contract Attacker {
    function () external payable {
        // Check state of the lottery before it's reset
        // or call back into it if logic allows
    }
}
```

2. Use this contract to enter the lottery and win
3. When `pickWinner()` sends funds, the fallback executes before `players = new address ;` Even though no reentrant call can be made here due to `.transfer`, teh pattern is unsafe and could introduce bugs in future changes, hence why it is only a medium-severity design flaw.

Recommended Mitigation: Update the state before any external calls:

```
// FIXED
address winner = players[index];
players = new address ;
winner.transfer(this.balance);
```

This follows the Checks-Effects-Interactions pattern, ensures the internal state is consistent before interacting with external contracts, and prepares the codebase for potential upgrades.

[M-2] Division by zero in winner selection

Description: The contract calculates the winner index using:

```
uint index = random() % players.length;
```

However, if `players.length == 0`, this line causes a division by zero error and the transaction will revert. This could happen if: - The manager mistakenly calls `pickWinner()` before any players have entered. - The contract is called maliciously to trigger a failure.

This is a standard logic flaw that should always be handled with a guard clause.

Impact: - Transaction reverts - Game becomes unplayable in edge cases - Blocks downstream calls if `pickWinner()` is used as part of a larger flow

While it does not result in loss of funds or reentrancy risk, it breaks contract logic and causes unnecessary failures.

Proof of Concept: 1. Deploy the contract

2. Call `pickWinner()` without any prior calls to `enter()`

```
await lottery.methods.pickWinner().send({ from: accounts[0] });
// Reverts with: division by zero
```

No protection is in place to ensure `players.length > 0`, so the call fails at runtime.

Recommended Mitigation: Add a simple check at the start of `pickWinner()`:

```
require(players.length > 0, "No players entered");
```

Example fix:

```
function pickWinner() public restricted {
    require(players.length > 0, "No players entered");
    uint index = random() % players.length;
    ...
}
```

This prevents accidental or malicious calls that would cause a revert due to empty state.

[M-3] No limit on number of entries per address, which can cause bias in winner selection.

Description: The `enter()` function allows the same address to call multiple times without restriction:

```
function enter() public payable {
    require(msg.value > .01 ether);
    players.push(msg.sender);
}
```

Impact: A single player could gain disproportionate odds, undermining the fairness of the lottery.

Proof of Concept: The following test was added to `Lottery.test.js`:

```
it('PoC: allows same address to enter multiple times', async () => {
    await lottery.methods.enter().send({
        from: accounts[1],
        value: web3.utils.toWei('0.02', 'ether')
    });

    await lottery.methods.enter().send({
        from: accounts[1],
        value: web3.utils.toWei('0.02', 'ether')
    });

    const players = await lottery.methods.getPlayers().call();
    console.log(players);
    assert.strictEqual(players.length, 2);
    assert.strictEqual(players[0], accounts[1]);
    assert.strictEqual(players[1], accounts[1]);
});
```

Running this test produced the following output:

```
[
  '0xeA528Bffe26a3385C7194B262277bf4343DeD255',
  '0xeA528Bffe26a3385C7194B262277bf4343DeD255'
]
  PoC: allows same address to enter multiple times
```

Recommended Mitigation: Enforce single-entry by tracking whether an address has already entered:

```
mapping(address => bool) public hasEntered;

function enter() public payable {
    require(msg.value > .01 ether);
    require(!hasEntered[msg.sender], "Already entered");
    players.push(msg.sender);
    hasEntered[msg.sender] = true;
}
```

[M-4] Unbounded Players Array Size

Description: The contract allows an unlimited number of players to enter the lottery:

```
address[] public players;

function enter() public payable {
    require(msg.value > .01 ether);
    players.push(msg.sender);
}
```

There is no cap on the size of the players array, and no mechanism to limit the number of entries per round. While the array is reset after `pickWinner()` is called, it can grow arbitrarily large between rounds.

This is particularly problematic because the `pickWinner()` function performs operations over the entire array.

Impact: - As `players[]` grows, the gas cost of `pickWinner()` increases - Eventually, `pickWinner()` may exceed the block gas limit, causing it to fail permanently - This can result in funds being locked in the contract unless an emergency mechanism is in place

In an on-chain lottery that accepts public entries, this creates a denial-of-service vector and limits scalability.

Proof of Concept: While the current implementation may work fine with a few dozen players, testing with thousands reveals gas issues:

```
for (let i = 0; i < 5000; i++) {
  await lottery.methods.enter().send({
    from: accounts[1],
    value: web3.utils.toWei("0.02", "ether")
  });
}

// Then try:
await lottery.methods.pickWinner().send({ from: accounts[0] });
// May fail due to out-of-gas
```

This simulates a real-world lottery experiencing high usage, resulting in gas exhaustion.

Recommended Mitigation: Set a reasonable upper bound on the number of players per round:

```
uint public maxPlayers = 100;

function enter() public payable {
    require(players.length < maxPlayers, "Player limit reached");
    require(msg.value > 0.01 ether);
    players.push(msg.sender);
}
```

This ensures: - Consistent and safe gas usage - Predictable transaction behavior - Better user experience during high load

Additionally, consider emitting an event when the cap is reached, or triggering `pickWinner()` automatically when the limit is hit.

Low

[L-1] Unrestricted Ether Contribution Amounts

Description: There is a minimum entry value of 0.01 ether, but no maximum. A user sending excessive ether receives only one slot in the lottery.

Impact: Creates fairness issues and may lead to disproportionate risk of loss for players.

Proof of Concept: Enter with 100 ether and compare against 0.02 ether entry, both get 1 slot.

Recommended Mitigation: Enforce maximum entry amounts, or scale entries by value contributed.

[L-2] Missing events for critical actions

Description: Key function (`enter`, `pickWinner`, reset) do not emit events.

Impact: Transparency and off-chain tracking of participation and winners is reduced.

Proof of Concept: Observe no logs for player joins or winner selection.

Recommended Mitigation: Emit events (e.g., `PlayersEntered`, `WinnerSelected`, `RoundReset`).

Informational

[I-1] Information Disclosure in `getPlayers()`

Description: `getPlayers()` publicly exposes the full list of entrants. While storage is already public on-chain, this makes participant addresses easily accessible.

Impact: May raise privacy concerns for players.

Proof of Concept: Call `getPlayers()` from any account; retrieve full participation list.

Recommended Mitigation: Consider restricting to manager or removing if privacy is desired.

[I-2] Outdated Compiler Version (^0.4.17)

Description: The contract specifies pragma solidity `^0.4.17`, which is an outdated compiler version with multiple known issues (see Solidity security advisories).

Impact: Projects compiled with outdated versions may be exposed to compiler-level bugs and lose compatibility with modern tooling.

Proof of Concept: Slither flagged `^0.4.17` as vulnerable to known historical compiler bugs.

Recommended Mitigation: Upgrade to at least Solidity 0.8.x and refactor for updated syntax and safety checks.

Gas

[G-1] Unnecessary dynamic array resets

Description: `players = new address` clears the array but doesn't refund storage as efficiently as possible.

Impact: Small gas inefficiency per round.

Proof of Concept: Observe gas cost difference between reallocating vs. using `delete players`.

Recommended Mitigation: Use `delete players` to clear array more efficiently.