UnluckySlug

NFT Lottery

Smart Contract Audit
Final Report

May 16, 2022

Introduction

1. About UnlukcySlug

Unlucky Slug is the world's first NFT lottery. It is a gambling system that allows any user to obtain a high-value NFT with a minimum investment. In addition, Unlucky slug is the first NFT club to have a lottery ticket, adding unique value to members who own a slug.

Visit www.unluckyslug.com to know more about it.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provides professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, and dydx.

The team has been able to secure 105+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-ups with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

The UnluckySlug team has provided the following doc for the purpose of audit:

1. https://github.com/solidityprog/unluckySlug#readme

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: UnluckSlug
- Contracts Name: UnluckySlug.sol
- Languages: Solidity(Smart contract), Typescript (Unit Testing)
- Github commits for initial audit: a542197519ee56f0233f8897a1983278a3bc1e0c
- Github commits for final audit: a2f2be55e64fa1584872d8218d7b6e10398e37e7
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level Reference

Every issue in this report were assigned a severity level from the following:

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	-	-	1
Closed	-	3	-

High Severity Issues

No issues were found.

Medium Severity Issues

M.1 Dos attack can be encountered due to out of GAS

Description: ***`UnluckySlug.withdrawAllNFTs`*** the external calls are made inside the for loop which can create a DOS situation.

Recommendation:
We recommend using the upper threshold in one for loop and doing external calls in that threshold.

Status: Acknowledged

M.2 State variables are written after the external call (Reentrancy)

Description: In all the below functions the state variable are written after making the external call which can create the Reentrancy scenario.

```
checkIfWinner, withdrawMediumNFT, withdrawNormalNFT, withdrawTopNFT, checkNFTPrize,
depositMediumNFT, depositNormalNFT, depositTopNFT, enterThrow, requestRandomWords, sendJackPot
```

Recommendation:
We recommend to follow CEI pattern to remove reentrancy or use Openzepplin ReentrancyGuard.

Status: Closed

M.3 Ignore return value of transfer

In UnluckySlug.withdrawERC20 the return value of the transfer is ignored. We recommend using SafeERC20 or ensuring that the transfer/transferFrom return value is checked.

Status: Closed

Low Severity Issues

L.1 Owner should be multisig

We recommend using a multisig account address (gnosis-safe) for the owner such that the single point of failure can't be achieved in the future.

Status: Acknowledged

L.2 No event emission for state change

Description
For most of the setter functions when the state is getting changed, there is no emission of event.

Recommendation:
We recommend emitting the event whenever the state gets changed.

Status: Open

Recommendation / Informational

I.1 Unnecessary use of safeMath wrappers

As 0.8 solc version onwards there is safeMath inbuilt so we recommend using an ***unchecked*** flag when safeMath wrappers are not needed.

for eg: In the enterThrow function by dividing by 100 we don't need any safeMath.

Status: Closed

I.2 Create constants for values not changed

Description: There are several values that need to be part of bytecode and should be constant as it's never been changed.

Recommendation:
Below variables can be made constant

```
VRF_COORDINATOR, callbackGasLimit, cashbackIncentivePercentage, numWords,
probabilityEquivalentToOne, referrerCommisionPercentage, requestConfirmations,
valuePercentageToJackpot
```

Status: Closed

I.3 Use literals and make constant

Description: The hardcoded values use be used as literals.

Recommendation:
The two variable `callbackGasLimit and LIMIT_GOLDEN_TICKETS` can be made constant and use literals for declaration

Status: Closed

I.4 Gas optimizations

- The below function should be made external from the public as by this activity the deployment gas will be reduced and bytecode of the contract too.
    - `setKeyHash, setSubscriptionID, setReferrer, setconstantProbability, setTicketCost, setJackPotProbability, enterThrow, depositFunds, depositNormalNFT, depositTopNFT, depositMediumNFT, withdrawAllNFTs, withdrawERC20, withdrawFunds, onERC721Received`
- The below line in **`checkNFTPrize`** can be omitted

```
if (nftRandomRange <= NFTarray[0]) {
            index = 0;
        }
```

As the index by default has a value of zero so the condition of reassigning it is a gas wastage.

Status: Closed

I.5 Used unlocked pragma

The pragma versions used in the contract are not locked. Consider
using the latest versions among 0.8.13 for deploying the contracts and libraries as it does not compile for any other version and can be confusing for a developer. Solidity source files indicate the versions of the compiler they can be compiled with.

pragma solidity ^0.8.0; // bad: compiles between 0.8.0 and 0.8.13
pragma solidity 0.8.0; // good : compiles w 0.8.0 only but not the latest version
pragma solidity 0.8.13; // best: compiles w 0.8.13

Status: Open

Unit Tests

No unit test cases were provided by the team.

Test cases must cover all function and imperative corner cases.

Automated Audit Result

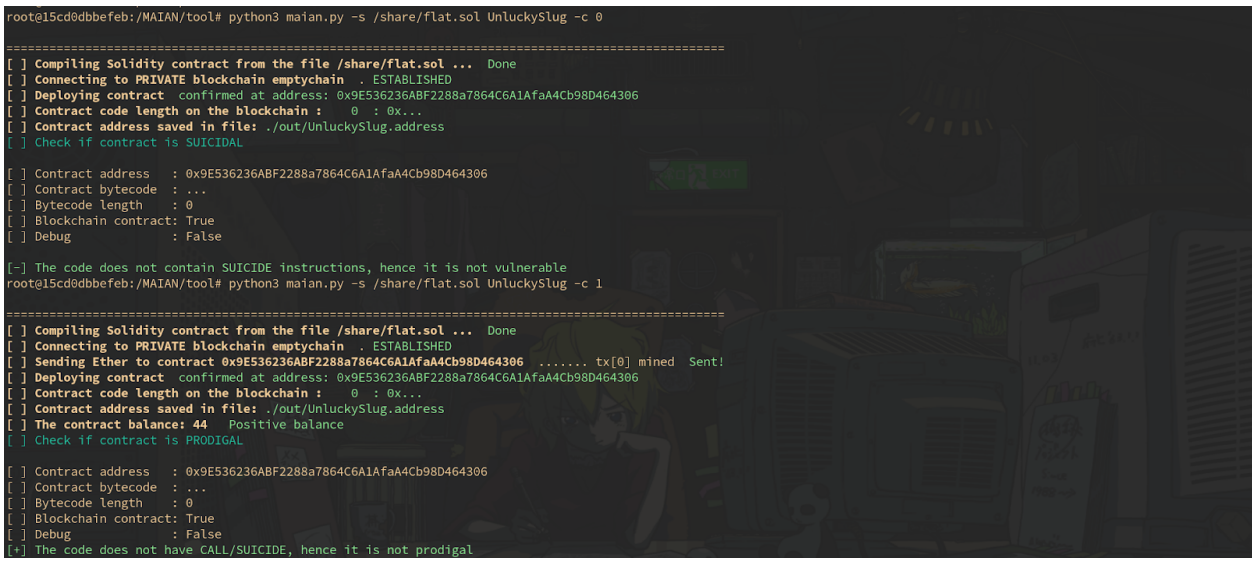
Slither

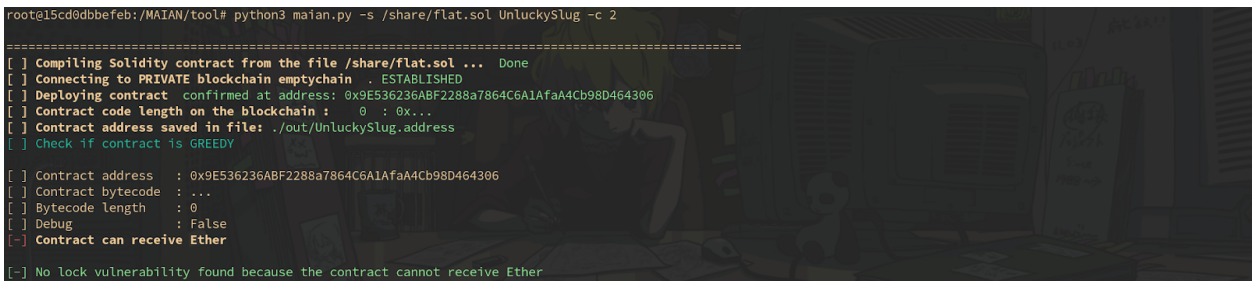
```
Compiled with solc
Number of lines: 2245 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 17 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 36
Number of informational issues: 68
Number of low issues: 26
Number of medium issues: 8
Number of high issues: 1
ERCs: ERC721, ERC20, ERC165

+-------------------------+-------------+---------------+--------------------+--------------+--------------------+
|          Name           | # functions |     ERCS      |     ERC20 info     | Complex code |      Features      |
+-------------------------+-------------+---------------+--------------------+--------------+--------------------+
| VRFCoordinatorV2Interface |      9      |               |                    |      No      |                    |
|          IERC20         |      6      |     ERC20     |     No Minting     |      No      |                    |
|                         |             |               | Approve Race Cond. |              |                    |
|                         |             |               |                    |              |                    |
|         Address         |     11      |               |                    |      No      |     Send ETH       |
|                         |             |               |                    |              |   Delegatecall     |
|                         |             |               |                    |              |     Assembly       |
|         Strings         |      4      |               |                    |     Yes      |                    |
|         Counters        |      4      |               |                    |      No      |                    |
|       UnluckySlug       |     93      | ERC165,ERC721 |                    |     Yes      |    Receive ETH     |
|                         |             |               |                    |              |     Send ETH       |
|                         |             |               |                    |              | Tokens interaction |
|                         |             |               |                    |              |     Assembly       |
+-------------------------+-------------+---------------+--------------------+--------------+--------------------+
flat.sol analyzed (17 contracts)
```

Mythril

```
The analysis was completed successfully. No issues were detected.
```

Maian





This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Concluding Remarks

While conducting the audits of the UnluckySlug smart contract, it was observed that the contracts contain Medium and Low severity issues.

Our auditors suggest that Medium and Low severity issues should be resolved by the developers. The recommendations given will improve the operations of the smart contract.

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the UnluckySlug platform or its product nor this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes