QuillAudits

Audit Report
May, 2023

For

ẹṣketit

Table of Content

Executive Summary

Project Name	Esketit
Overview	Esketit and EsketitMarketplace contracts are designed for minting, burning, and trading of ERC721 tokens. The Openzeppelin library was also integrated in contract to help with ownable, non-reentrant and pausable features. Only Esketit users can acquire NFT tokens from the marketplace contract.
Timeline	23 MArch, 2023 - 31 March, 2023
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyse Esketit's Esketit and EsketitMarketplace codebases for quality,security, and correctness. Commit hash: c7241e1448f5e2313421e959cf05775f909839cb
Fixed In	6160ab636d70fe305c5a0f50a88e7bd12609f3e2

High
Medium
Low
Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	1
Partially Resolved Issues	0	0	0	0
Resolved Issues	1	2	1	3

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Contract - Common Issues

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

Informational Issues

A.1 Floating Solidity (pragma solidity ^0.8.0)

Description

Both contracts have a floating solidity pragma version. This is also present in inherited contracts. Locking the pragma helps to ensure that the contract does not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively. The recent solidity pragma version also possesses its own unique bugs.

Remediation

Making the contract use a stable solidity pragma version prevents bugs that could be ushered in by prospective versions. Using a fixed solidity pragma version while deploying is recommended to avoid deployment with versions that could expose the contract to attack.

Status

Resolved

A.2 Use != 0 Instead of > 0 in Order to Reduce Gas Cost

The differences in gas when these functions are called is 44 gas. This implies that using the operations that ensure that the operand is not equal to zero is cheaper.

Esketit.sol

```
ftrace | funcSig
70  function updatePricePerItem(uint256 _price⬆) external onlyOwner {
71      require(_price⬆ > 0, "zero price");      You, 5 days ago • chore:
72      _pricePerItem = _price⬆;
73      emit UpdatePricePerItem(_pricePerItem);
74  }
```

EsketitMarketplace.sol

```
100
101 ▌   modifier isListed(address _nftAddress⬆, uint256 _tokenId⬆, address _owner⬆) {
102         Listing memory listing = listings[_nftAddress⬆][_tokenId⬆][_owner⬆];
103         require(listing.price > 0, "not listed item");      You, 5 days ago • chore:
104         _;
105     }
106
```

Recommendation

check that the parameters that serve as operands are not equal to 0 rather than them being greater than. The code snippet below was used in contract and it is same as the recommendation provided.

```
178
179     /// Change max purchased limit.
        ftrace | funcSig
180     function updateMaxBuyLimit(uint16 _maxLimit⬆) external onlyOwner {
181         require(_maxLimit⬆ != 0, "can't set to zero");      You, 5 days
182         _maxBuyLimit = _maxLimit⬆;
183         emit UpdateMaxBuyLimit(_maxBuyLimit);
184     }
185
```

Status

Resolved

A.2 Insufficient Code Comments

Description

The code comments currently are not sufficient Good code comments are relevant for easy comprehension of the contract; it aids users and developers interacting with the contract to duly understand the motive behind every function in the contract.

Remediation

It is recommended to add comments to every existing function in the contract. The Natspec format is a well-detailed format that will aid in achieving good comment and documentation.

Status

Acknowledged

B. Esketit.sol

High Severity Issues

No issues found

Medium Severity Issues

B.1 Possibility of Tokens Getting Stuck in Contract When Updating Marketplace Address

Description

With the _setApprovalForAll function called at the constructor level, the esketitMarketplace contract becomes the only address that can transfer out tokens to buyers because these tokens are minted directly into the token contract itself. However, issue of stuck tokens arises when contract owner updates the esketitMarketplace variable to a new marketplace address, tokens in the token contract cannot be called by these new marketplace address because approval was only set at the deployment time and the approval won't persist for new esketitMarketplace variables.

Remediation

Call the _setApprovalForAll function when updating the marketplace address.

Status

Resolved

Low Severity Issues

No issues found

Informational Issues

No issues found

C. EsketitMarketplace.sol

High Severity Issues

C.1 Malicious Owner Can Buy Tokens From Esketit User with Zero Amount or Price Less Than the Allocated Price Per Item

```
160     function esketitBuyBack(
161         uint256[] calldata _tokenId⬆,
162         uint256[] calldata _price⬆,
163         address _nftAddress⬆,
164         address _payToken⬆,
165         address _esketitUser⬆
166     ) external onlyOwner {
167         uint256 _totalPrice;
168
169         for (uint256 itr = 0; itr < _tokenId⬆.length; itr++) {
170             IERC721(_nftAddress⬆).safeTransferFrom(_esketitUser⬆, _nftAddress⬆, _tokenId⬆[itr]);
171             _totalPrice += _price⬆[itr];
172         }
173
174         IERC20(_payToken⬆).safeTransferFrom(esketitTreasury, _esketitUser⬆, _totalPrice);
175
176         emit EsketitBuy(_nftAddress⬆, _esketitUser⬆, _totalPrice, _payToken⬆);
177     }       You, 5 days ago • chore: forge init
```

The esketitBuyBack function allows the owner to buy back tokens from esketit users who have given the marketplace an approval. This function allows for the owner to manually input the prices to pay from the esketitTreasury address. A malicious owner can exploit this function by providing zero value or amount less than the pricePerItem variable meant to serve as the cost of the tokens.

Recommendation

It is recommended to call the pricePerItem function from the token contract and use this value as the _totalPrice to send to the sellers.

Status

Resolved

Medium Severity Issues

C.2 Users are Unable to List Items Due to Invalid Check

```
198
199     /// List NFT on secondary marketplace
        ftrace | funcSig
200     function listItem(address _nftAddress↑, uint256 _tokenId↑, address _payToken↑, uint256 _price↑, uint256 _startingTime↑)
201         external
202         notListed(_nftAddress↑, _tokenId↑, _msgSender())
203     {
204         require(esketitUser[_msgSender()], "unknown user");
205         require(esketitLoanBasket[_msgSender()], "unknown nft address");
206         require(payTokens[_payToken↑], "not verified payToken");
207
208         IERC721 nft = IERC721(_nftAddress↑);
209
210         require(nft.ownerOf(_tokenId↑) == _msgSender(), "not owning item");
211         require(nft.isApprovedForAll(_msgSender(), address(this)), "item not approved");
212
213         listings[_nftAddress↑][_tokenId↑][_msgSender()] = Listing(_payToken↑, _price↑, _startingTime↑);
214         emit ItemListed(_msgSender(), _nftAddress↑, _tokenId↑, _payToken↑, _price↑, _startingTime↑);
215     }
```

The marketplace allows users to list nft tokens that are verified by the platform owner. This is ascertained with the esketitLoanBasket mapping that serves verified nft addresses to true. In the listItem function, instead of a check that verifies the provided _nftAddress variable, the check instead checks if the msg.sender address is true. This will cause the function call to revert and deny users the list item functionality.

Recommendation

redesign the contract to check for the _nftAddress variables in the esketitLoanBasket instead of the address of the caller.

Status

Resolved

Low Severity Issues

C.3 Input Validation

```
279     /// Change the fee recipient wallet address.
        ftrace | funcSig
280     function updateFeeRecipient(address payable _feeRecipient⬆) external onlyOwner {
281         feeRecipient = _feeRecipient⬆;
282         emit UpdatePlatformFeeRecipient(_feeRecipient⬆);
283     }
284
285     /// Change the esketit treasury wallet address.
        ftrace | funcSig
286     function updateEsketitTreasury(address payable _esketitTreasury⬆) external onlyOwner {
287         esketitTreasury = _esketitTreasury⬆;
288         emit UpdateEsketitTreasury(esketitTreasury);
289     }
290
291     /// Change the verified payToken status.
        ftrace | funcSig
292     function updatePayToken(address _payToken⬆, bool _status⬆) external onlyOwner {
293         require(_payToken⬆ != address(0), "zero address");
294         payTokens[_payToken⬆] = _status⬆;
295         emit UpdatePayToken(_payToken⬆, _status⬆);
296     }
```

Zero addresses can be passed in without reverting, and any arbitrary amount can go in for platformFee as well. It is necessary to sanitize input to avoid breaks in code because of unexpected values.

Recommendation

Before critical changes to the contract state, checks can be made; for example: to confirm the previous value is not the same as the new one to be added, zero address checks, integer over or underflows. These checks should happen at the beginning of the functions to ensure reverts happen before excessive gas is consumed.

Status

Resolved

Informational Issues

C.4 Esketit Token Contract Fails to Implement ERC721 Receiver Causing the Contract to Revert

Description

When a contract owner buys from Esketit Users via the esketitBuyBack function, the tokens are sent to an NFT address. If the contract address fails to implement the ERC721 receiver, the function will revert.

Remediation

Integrate ERC721 receiver into the token contract so that it can receive nft tokens bought from Esketitusers.

Status

Resolved

C.5 Gas Cost Increase Due to Struct not Tightly Packed

```
56      /// Structure for listed items
        ...
57      struct Listing {
58          address payToken;
59          uint256 price;
60          uint256 startingTime;
61      }       You, 5 days ago • chore: forge init
```

The current struct takes 3 slots and can be tight packed to help reduce gas consumption.

Remediation

This can be reduced to two slots that wraps address payToken and uint96 startingTime to one slot and then the price takes another slot.

Reference

Status

Resolved

D.4 General Recommendation

Both contracts have some privilege functions that can only be called by the contract owner. Adding users, adding approved tokens for payment, and adding verified nft address into the esketitLoanBasket are only owner functions. It is advised that these important state variables are initialized in order to let all functions work appropriately.

Functional Testing

Some of the tests performed are mentioned below

- Should get the name of the token
- Should get the symbol of the token
- Should revert when nft tokens are sent to the token contract
- Should burn tokens when given an approval by original users
- Should pause and unpause the transfer of tokens by approved persons
- Should successfully buy from the marketplace using the primary market
- Should successfully buy from esketit users who approves marketplace address
- Should list items of verified nfts on the marketplace.
- Should successfully buy items formerly listed on the marketplace by a user.
- Should update critical state changes like platform fee, users, paytokens and verified nft addresses.
- Should reverts when unverified nft address is to be listed on the platform
- Should revert when buying items that have not started.
- Should revert when buying canceled listed items.
- Should revert when buying non-existing items from the marketplace.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of Esketit. We performed our audit according to the procedure described above.

Some issues of high, medium, low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Esketit Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Esketit Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

700+
Audits Completed

$16B
Secured

700K
Lines of Code Audited

Follow Our Journey

Audit Report
May, 2023

For

esketit

QuillAudits

Canada, India, Singapore, United Kingdom

www.quillaudits.com

audits@quillhash.com