SMART CONTRACT AUDIT REPORT

for

RosePad

Prepared By: Xiaomi Huang

PeckShield

August 16, 2022

Document Properties

Client	RosePad
Title	Smart Contract Audit Report
Target	RosePad
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 16, 2022	Luck Hu	Final Release
1.0-rc2	August 10, 2022	Luck Hu	Release Candidate #2
1.0-rc1	July 30, 2022	Luck Hu	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the RosePad protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About RosePad

RosePad is a premium launchpad with DEX and NFTs on Oasis Network. It aims to be the go-to platform for fundraising, trading and connecting to Oasis community. The goal of RosePad is to provide a platform to launch hand-picked projects on Oasis Network and bring real impact and value to the communities while contributing to adoption of Oasis blockchain. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The RosePad

Item	Description
Issuer	RosePad
Website	https://rosepad.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 16, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audit scope only covers the following smart contracts: staking/RosePadNFTStakingPoints.sol, staking/RosePadRewardContainer.sol, staking/RosePadStakeMaster.sol, staking/RosePadStakeVault.sol, staking/RosePadStakeVaultFactory.sol, swap/RosePadSwapFactory.sol, swap/RosePadSwapLP.sol, swap/RosePadSwapPair.sol, swap/RosePadSwapRouter.sol, tokens/RoseApe721.sol, tokens/RosePadToken.sol, utils/Lockable.sol, utils/RosePadNFTRarityDict.sol, and their dependent interfaces in /interfaces folder.

- https://github.com/rosepad-tech/launchpad-contracts.git (b059c0e)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/rosepad-tech/launchpad-contracts.git (5a4dd3e)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;

- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `RosePad` protocol implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■ ■
Low	2	■ ■
Informational	0	
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key RosePad Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Sanity Checks For System/-Function Parameters	Coding Practices	Confirmed
PVE-002	Low	Possible Bypass of BOOSTER_-LOCK_DURATION And Owner-shipLimit	Business Logic	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Low	Proper Token Ownership Transfer in RoseApe721	Business Logic	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improved Sanity Checks For System/Function Parameters

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: RoseApe721
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The RoseApe721 contract implements the NFT token for the RosePad. It defines a state variable (uint256 public _maxSupply = 5555) which gives the maximum amount of RoseApes NFTs that can be minted. While examining the logic to ensure the _maxSupply can not be exceeded, we observe the need to revisit the current implementation.

To elaborate, we show below the code snippet of the mint() routine. At the beginning of the routine, it validates the current _tokenIdCounter to not exceed the defined _maxSupply, or it directly reverts the mint. However, the current validation is insufficient in allowing more RoseApes NFTs to be minted than the _maxSupply, because it does not take the qty of new RoseApes NFTs to be minted into consideration. With that, we suggest to tighten the sanity check (line 58) as: require(_tokenIdCounter .current()+ qty < _maxSupply + 1).

```
54      uint256 public _maxSupply = 5555;
55      function mint(uint256 qty) public payable virtual {
56      require(!paused);
57      require(msg.value >= 0, "Not enough ROSE sent; check price!");
58      require(_tokenIdCounter.current() < _maxSupply);
59
60      //  Whitelist mode. We will have a whitelist event.
61      if (whitelistMode) {
62          if (whitelistCheck) {
63              require(whitelisted[msg.sender], "Only whitelist participants are allowed
                    during whitelist sale.");
64          }
```

```
        uint256 requiredAmount = qty * _whitelistSalePrice;
        uint256 arrayLength = userOwnedTokensWhitelist[msg.sender].length;
        uint256 toBeTotal = arrayLength + qty;
        require(toBeTotal < (_whitelistOwnershipLimit + 1), "Maximum Holding for WL
            Event"); // only 3 allowed!
        require(msg.value >= requiredAmount, "Not enough ROSE sent; check price!");

        // Mint for whitelist
        for (uint256 i = 1; i <= qty; i++) {
            _tokenIdCounter.increment();
            uint256 tokenId = _tokenIdCounter.current();
            userOwnedTokensWhitelist[msg.sender].push(tokenId);
            _mint(msg.sender, tokenId);
        }
    } else {
        uint256 requiredAmount = qty * _publicSalePrice;
        uint256 arrayLength = userOwnedTokensPublic[msg.sender].length;
        uint256 toBeTotal = arrayLength + qty;
        require(toBeTotal < (_publicOwnershipLimit + 1), "Maximum Holding for Public
            Event"); // only 15 allowed!
        require(msg.value >= requiredAmount, "Not enough ROSE sent; check price!");

        // Mint for public
        for (uint256 i = 1; i <= qty; i++) {
            _tokenIdCounter.increment();
            uint256 tokenId = _tokenIdCounter.current();
            userOwnedTokensPublic[msg.sender].push(tokenId);
            _mint(msg.sender, tokenId);
        }
    }
}
```

Listing 3.1: `RoseApe721::mint()`

Recommendation Revisit the above mentioned routine to add proper sanity checks.

Status This issue has been confirmed by the team that this contract has already been launched, so they would like to keep it.

3.2 Possible Bypass of BOOSTER_LOCK_DURATION And OwnershipLimit

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RosePadStakeMaster`, `RoseApe721`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `RosePadStakeMaster` contract provides an incentive mechanism that rewards the staking of supported assets (`RDPA`, `LP`s and `RoseApes NFT`s) with the reward token in the `rewardContainer`. The stakes can be withdrawn in the `withdraw()` routine. Specially, the `RoseApes NFT`s can be withdrawn separately in the `withdrawAllNFTs()` routine.

To elaborate, we show below the code snippets of the `withdraw()/withdrawAllNFTs()` routines. In the `withdrawAllNFTs()`, the deposited `NFT`s are locked for a duration of `BOOSTER_LOCK_DURATION` (line 278) before they can be withdrawn. However, in the `withdraw()`, the deposited assets are locked for a duration of `user.lockEndTime` (line 224) which is provided by the stake holder in the `_deposit()`. Apparently, the `user.lockEndTime` may be different with the `BOOSTER_LOCK_DURATION`. As a result, if users deposit `NFT`s only, they can withdraw them after inconsistent lock durations.

```
275     function withdrawAllNFTs() external nonReentrant whenNotPaused {
276         UserInfo storage user = userInfo[msg.sender];
277         require(user.lastNftDepositTime > 0, "withdrawAllNFTs: NO_BOOSTER_LOCKED");
278         require(user.lastNftDepositTime + BOOSTER_LOCK_DURATION < block.timestamp, "
            withdrawAllNFTs: BOOSTER_LOCKED");
279
280         // harvest
281         _harvest(msg.sender);
282
283         for (uint256 i = 0; i < user.nftIds.length; i++) {
284             uint256 nftId = user.nftIds[i];
285             if (user.basisPointsOfNFT[nftId] > 0) {
286                 user.basisPointsOfNFT[nftId] = 0;
287             }
288             if (user.fixedPointsOfNFT[nftId] > 0) {
289                 user.fixedPointsOfNFT[nftId] = 0;
290             }
291             nft.safeTransferFrom(address(this), msg.sender, nftId);
292         }
293         emit WithdrawAllNFTs(msg.sender, user.nftIds);
294
295         delete user.boosterBasisPoints;
296         delete user.boosterFixedPoints;
297         delete user.nftIds;
298     }
```

Listing 3.2: RosePadStakeMaster:withdrawAllNFTs()

```
221     function withdraw() external nonReentrant whenNotPaused {
222         UserInfo storage user = userInfo[msg.sender];
223         require(user.rpadAmount > 0  user.lpTokenAmount > 0  user.nftIds.length > 0, "
            withdraw: NOTHING_LOCKED");
224         require(user.lockEndTime < block.timestamp, "withdraw: LOCKED");
225         ...
226         // Transfer all NFTs to user
227         for (uint256 i = 0; i < user.nftIds.length; i++) {
```

```
            uint256 nftId = user.nftIds[i];
            if (user.basisPointsOfNFT[nftId] > 0) {
                user.basisPointsOfNFT[nftId] = 0;
            }
            if (user.fixedPointsOfNFT[nftId] > 0) {
                user.fixedPointsOfNFT[nftId] = 0;
            }
            nft.safeTransferFrom(address(this), msg.sender, nftId);
        }
        emit Withdraw(msg.sender, user.rpadAmount, user.lpTokenAmount, user.nftIds);

        delete user.rpadAmount;
        delete user.lpTokenAmount;
        delete user.nftIds;
        delete user.boosterBasisPoints;
        delete user.boosterFixedPoints;
        delete user.lockStartTime;
        delete user.lockEndTime;
    }
```

Listing 3.3: RosePadStakeMaster:withdraw()

What is more, the `RoseApe721` contract provides a `mint()` routine for `RoseApes NFTs` selling. Specially, it defines two state variables (`_whitelistOwnershipLimit` and `_publicOwnershipLimit`) that represent the maximum amounts of `RoseApes NFTs` one user can hold for whitelist mode and public mode. While examining the validation of the two maximum holding limits, we notice they can be potentially bypassed.

In the following, we show the code snippet of the `mint()` function. If we consider the public mode (`!whitelistMode`), this function counts the new total amount of `RoseApes NFTs` (`toBeTotal`) the `msg.sender` can hold after the public mint and validates the `toBeTotal` to be no more than the `_publicOwnershipLimit` (line 81). However, we notice that malicious user can create a lot of temporary contracts to mint `RoseApes NFTs` and then transfer all of them to the user. As a result, the user can hold more `RoseApes NFTs` than the `_publicOwnershipLimit` from public mint. Based on this, it is suggested to transfer the token ownership accordingly for token transfer.

```
54    function mint(uint256 qty) public payable virtual {
55        require(!paused);
56        require(msg.value >= 0, "Not enough ROSE sent; check price!");
57        require(_tokenIdCounter.current() < _maxSupply);
58
59        // Whitelist mode. We will have a whitelist event.
60        if (whitelistMode) {
61            if (whitelistCheck) {
62                require(whitelisted[msg.sender], "Only whitelist participants are
                      allowed during whitelist sale.");
63            }
64            uint256 requiredAmount = qty * _whitelistSalePrice;
65            uint256 arrayLength = userOwnedTokensWhitelist[msg.sender].length;
66            uint256 toBeTotal = arrayLength + qty;
```

```
            require(toBeTotal < (_whitelistOwnershipLimit + 1), "Maximum Holding for WL Event"); // only 3 allowed!
            require(msg.value >= requiredAmount, "Not enough ROSE sent; check price!");

            //  Mint for whitelist
            for (uint256 i = 1; i <= qty; i++) {
                _tokenIdCounter.increment();
                uint256 tokenId = _tokenIdCounter.current();
                userOwnedTokensWhitelist[msg.sender].push(tokenId);
                _mint(msg.sender, tokenId);
            }
        } else {
            uint256 requiredAmount = qty * _publicSalePrice;
            uint256 arrayLength = userOwnedTokensPublic[msg.sender].length;
            uint256 toBeTotal = arrayLength + qty;
            require(toBeTotal < (_publicOwnershipLimit + 1), "Maximum Holding for Public Event"); // only 15 allowed!
            require(msg.value >= requiredAmount, "Not enough ROSE sent; check price!");

            //  Mint for public
            for (uint256 i = 1; i <= qty; i++) {
                _tokenIdCounter.increment();
                uint256 tokenId = _tokenIdCounter.current();
                userOwnedTokensPublic[msg.sender].push(tokenId);
                _mint(msg.sender, tokenId);
            }
        }
    }
```

Listing 3.4: RoseApe721:mint()

Recommendation Revisit the above mentioned routines to properly tighten the validation of `BOOSTER_LOCK_DURATION` and `OwnershipLimit`.

Status The "Bypass of BOOSTER_LOCK_DURATION" issue has been fixed in this commit: `5a4dd3e`, and the "Bypass of NFT OwnershipLimit" issue is confirmed by the team that the contract has already been launched, so they would like to keep it.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the RosePad protocol, there is a privileged account, i.e., owner, that plays a critical role in governing and regulating the system-wide operations (e.g., mint new RoseApes NFTs). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the RoseApe721 contract as an example and show the representative functions potentially affected by the privileges of the owner account.

Specifically, the privileged function in RoseApe721 allows for the owner to mint new RoseApes NFTs to the to address.

```
    function safeMint(address to) public onlyOwner {
        require(_tokenIdCounter.current() < _maxSupply);

        // increment
        _tokenIdCounter.increment();
        uint256 tokenId = _tokenIdCounter.current();
        _safeMint(to, tokenId + 1);
    }
```

Listing 3.5: Example Privileged Operations in the RoseApe721 Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged owner account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated by the team and they have renounced the ownership of the NFT contract.

3.4 Proper Token Ownership Transfer in RoseApe721

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RoseApe721
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `RoseApe721` contract defines two state variables that maintain the mappings from the token holder to all the token ids he/she holds from the public mint and the whitelist mode mint. Our analysis shows the current approach of token ownership transfer can be improved.

To elaborate, we show below the code snippets of the `mint()` routine. Let us take the public mint (`!whitelistMode`) for example, when the new `tokenId` is minted to the user, the routine pushes the `tokenId` to the `userOwnedTokensPublic[msg.sender]` (line 88). While examining the token transfer/burn logic, we notice there is a lack of properly transferring the ownership from the old holder to the new holder. Based on this, it is suggested to properly transfer the ownership accordingly during token transfer/burn.

```
54     function mint(uint256 qty) public payable virtual {
55         require(!paused);
56         require(msg.value >= 0, "Not enough ROSE sent; check price!");
57         require(_tokenIdCounter.current() < _maxSupply);
58 
59         //  Whitelist mode. We will have a whitelist event.
60         if (whitelistMode) {
61             if (whitelistCheck) {
62                 require(whitelisted[msg.sender], "Only whitelist participants are
                       allowed during whitelist sale.");
63             }
64             uint256 requiredAmount = qty * _whitelistSalePrice;
65             uint256 arrayLength = userOwnedTokensWhitelist[msg.sender].length;
66             uint256 toBeTotal = arrayLength + qty;
67             require(toBeTotal < (_whitelistOwnershipLimit + 1), "Maximum Holding for WL
                   Event"); // only 3 allowed!
68             require(msg.value >= requiredAmount, "Not enough ROSE sent; check price!");
69 
70             //  Mint for whitelist
71             for (uint256 i = 1; i <= qty; i++) {
72                 _tokenIdCounter.increment();
73                 uint256 tokenId = _tokenIdCounter.current();
74                 userOwnedTokensWhitelist[msg.sender].push(tokenId);
75                 _mint(msg.sender, tokenId);
76             }
77         } else {
78             uint256 requiredAmount = qty * _publicSalePrice;
79             uint256 arrayLength = userOwnedTokensPublic[msg.sender].length;
80             uint256 toBeTotal = arrayLength + qty;
81             require(toBeTotal < (_publicOwnershipLimit + 1), "Maximum Holding for Public
                    Event"); // only 15 allowed!
82             require(msg.value >= requiredAmount, "Not enough ROSE sent; check price!");
83 
84             //  Mint for public
85             for (uint256 i = 1; i <= qty; i++) {
86                 _tokenIdCounter.increment();
87                 uint256 tokenId = _tokenIdCounter.current();
```

```
                userOwnedTokensPublic[msg.sender].push(tokenId);
                _mint(msg.sender, tokenId);
            }
        }
    }
```

Listing 3.6: RoseApe721:mint()

Recommendation Revisit the token transfer/burn logic to properly transfer/burn the ownership accordingly.

Status This issue has been confirmed by the team that this contract has already been launched, so they would like to keep it.

4 Conclusion

In this audit, we have analyzed the RosePad design and implementation. RosePad is a premium launchpad with DEX and NFTs on Oasis Network. It is a go-to platform for fundraising, trading and connecting to Oasis community. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.