Smart Contract Audit Report

NFTSTAR Smart Contract

6 June 2022

Numen Cyber Labs - Security Services

Numen Cyber Technology Pte. Ltd.
11 North Buona Vista Drive, #04-09,
The Metropolis, Singapore 138589

Tel:	65-63555555
Fax:	65-63666666
Email:	sales@numencyber.com
Web:	https://numencyber.com

Table of Content

1 Executive Summary

Numen Cyber Technology was engaged by NFTSTART to review smart contract implementation. The assessment was conducted in accordance with our systematic approach to evaluate potential security issues based on customer requirements. The report provides detailed recommendations to resolve the issue and also provides additional suggestions or recommendations for improvement.

A High severity finding was identified in no-transaction-fee mining and the inability to create new transactions when large transactions were involved.

The outcome of the assessment outlined in chapter 3 provides the system's owners with a full description of the vulnerabilities identified, the associated risk rating for each vulnerability, and detailed recommendations that will resolve the underlying technical issue.

1.1 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10] which is the gold standard in risk assessment using the following risk models:

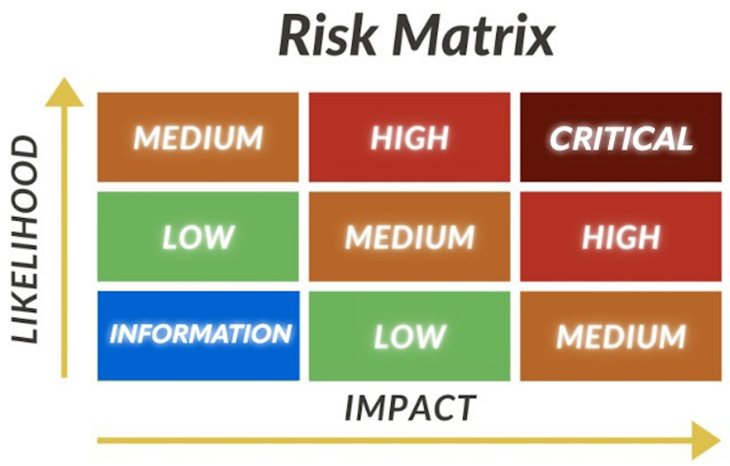
• Likelihood: represents how likely a particular vulnerability is to be uncovered and exploited in the wild.

• Impact: measures the technical loss and business damage of a successful attack;

• Severity: determine the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: High, Medium, and Low. Severity is determined by likelihood and impact and can be classified into four categories accordingly, Critical, High, Medium, Low shown in table 1.1.

Table 1.1: Overall Risk Severity



To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. The audit was performed in a systematic approach guided by a comprehensive assessment list carefully designed that targets known and impactful security issues. If our tool or analysis does not identify any issue, the contract is considered safe regarding the assessment item. For any discovered issue, we might further deploy contracts on our private test environment 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.

- 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 our tool finds.
- Code and business security testing: We further review business logic, 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.

Table 1.2: The Full List of Assessment Items

Category	Assessment Item
Basic Coding Assessment	Apply Verification Control
	Authorization Access Control
	Forged Transfer Vulnerability
	Forged Transfer Notification
	Numeric Overflow
	Transaction Rollback Attack
	Transaction Block Stuffing Attack
	Soft_fail Attack
	Hard_fail Attack
	Abnormal Memo

	Abnormal Resource Consumption
	Secure Random Number
Advanced Source Code Scrutiny	Asset Security
	Cryptography Security
	Business Logic Review
	Source Code Functional Verification
	Account Authorization Control
	Sensitive Information Disclosure
	Circuit Breaker
	Blacklist Control
	System API Call Analysis
	Contract Deployment Consistency Check
Additional Recommendations	Semantic Consistency Checks
	Following Other Best Practices

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [14], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development.

1.2 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the non-existence 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 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 Overview

2.1 Summary

Severity	**# of Findings**	
Critical	0	

High	3	🟥🟥🟥
Medium	0	
Low	2	🟩🟩
Informational	4	🟦🟦🟦🟦
Total	9	🟥🟥🟥🟩🟩🟦🟦🟦🟦

2.2 Key Findings

Three high severities findings identified relate to the omission of fee calculation in currency mining. In addition, two are Low, and four are Informational.

Table 2.1: Key Audit Findings

ID	**Severity**	**Findings Title**	**Status**
NVE-001	High	Incorrect Setting of MaxAmount and MaxLimit	
NVE-002	High	Improper Setting of Opening flag	
NVE-003	High	Risks in the authority transfer mechanism	
NVE-004	Low	grantLimits Function is not compared with the Total Supply of the token	
NVE-005	Low	giftLimit variable declared but not used	
NVE-006	Informational	Multiple Compiler Versions Declared	
NVE-007	Informational	Improper Usage Of Public Functions	
NVE-008	Informational	Missing Emit Events	
NVE-009	Informational	Optimization of Variable Type	

3 Detailed Description of Findings

3.1 Incorrect Setting of MaxAmount and MaxLimit

ID: NVE-001
Severity: High
Likelihood: Meduim
Impact: High

Location: CrowdSale.sol
Category: Business Issues

Description

setMaxAmount should be smaller than TOTAL_SUPPLY and *SetMaxLimit* should be smaller than the max value, if the max value is bigger then this will potentially lead to mint amount exceeding TOTAL_SUPPLY by call *preMint* and *pubMint* in the first-time call.
This is because function *preMint* and *pubMint* only compare the parameter _amount with the limit and in the first time calling, token.current() is just 0, not updated yet, so the code *require(token.current() <= TOTAL_SUPPLY, "Exceeded total supply");* will be executed successfully without any exception and if attacker get the owner privilege or the project party itself has set a value greater than total_supply by mistake, which will result in minting more than the total number of tokens:

```
function setMaxAmount(uint32 _amount)
    external
    onlyOwner
    onlyPositive(_amount)
{

    max = _amount;
    event SettedMaxAmount(max);
}

function setMaxLimit(uint32 _limit)
        external
        onlyOwner
        onlyPositive(_limit)
    {
        limit = _limit;
    }
```

Exploitation Steps:
Condition: the attacker gets the contract owner's private key, or the project set the limit exceeding the TOTAL_SUPPLY by error.
1.Call *setMaxLimit* function with the value bigger than TOTAL_SUPPLY e.g:1000

2.Call *setOpening* function and the parameter set to true when next step call *preMint* or *setClosing* function and parameter set to true

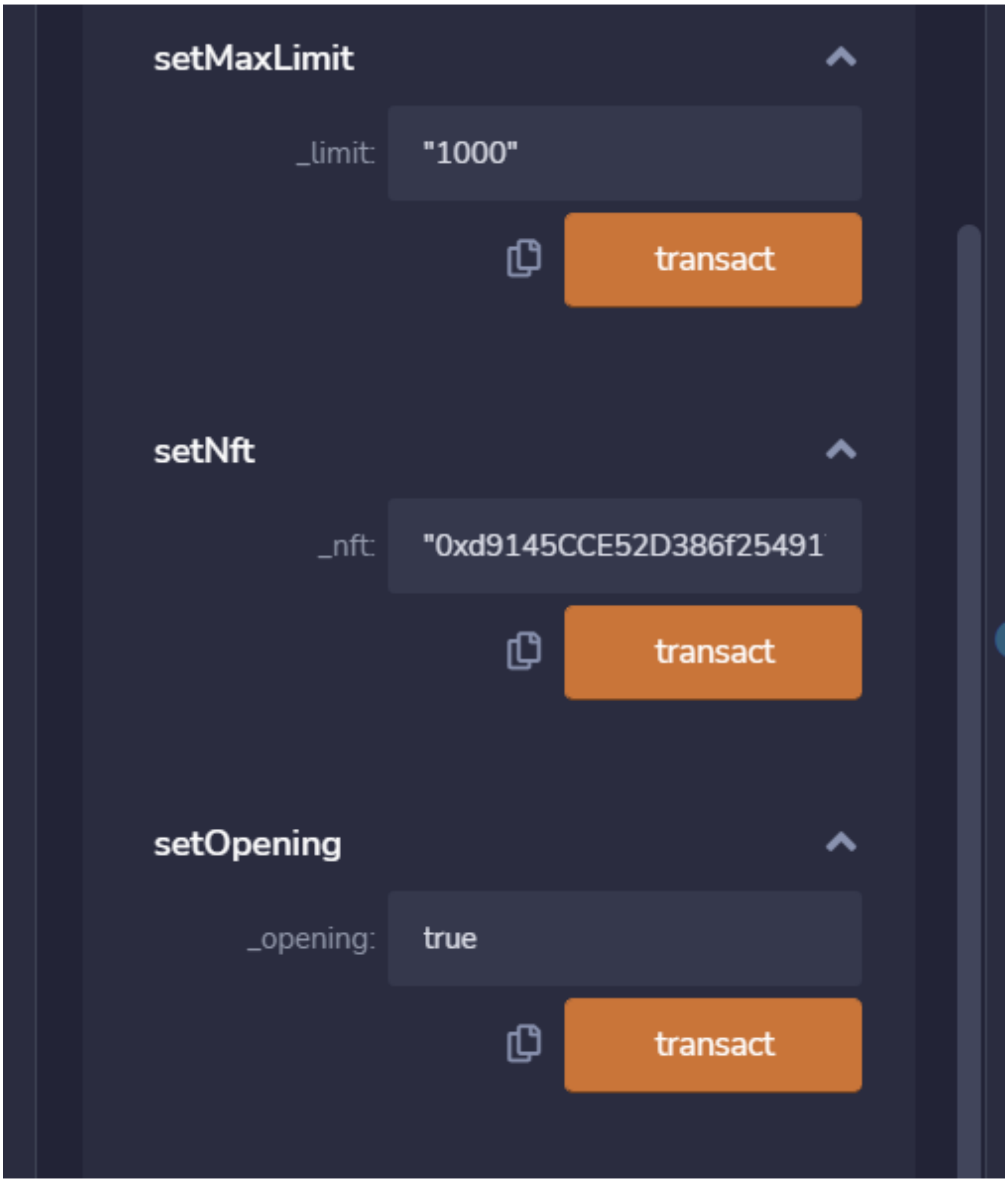
3.Call *preMint* function with _amount =1000 or pubMint with _amount =1000

```
    function preMint(uint256 _amount)
        external
        payable
        onlyPositive(_amount)
    {
        require(opening, "PreSales time has not started");
        require(_amount <= limit, "More than one purchase");
        //require(msg.value == _amount.mul(preSalePrice), "Payment declined");
        require(token.current() <= TOTAL_SUPPLY, "Exceeded total supply");
        address miner = msg.sender;
        // require(hasRole(MINER_ROLE, miner), "Address not whitelisted");
        sold[miner] = _amount.add(sold[miner]);
        // (bool ok, ) = quotas[miner].trySub(sold[miner]);
        // require(ok, "Exceeds Allocation");
        _asyncTransfer(collector, msg.value);
        token.mint(miner, _amount);
    }

    function pubMint(uint256 _amount) external payable onlyPositive(_amount) {
        require(closing, "PubSales time has not started");
        require(_amount <= limit, "More than one purchase");
        //require(msg.value == _amount.mul(publicSalePrice), "Payment declined");
        require(token.current() <= TOTAL_SUPPLY, "Exceeded total supply");
        address miner = msg.sender;
        sold[miner] = _amount.add(sold[miner]);
        // (bool ok, ) = max.trySub(sold[miner]);
        // require(ok, "Exceeded maximum quantity limit");
        //_asyncTransfer(collector, msg.value);
        token.mint(miner, _amount);
    }
```

The following provides the evidence of exploiting the issue:

1. Set the erroneous Limit.



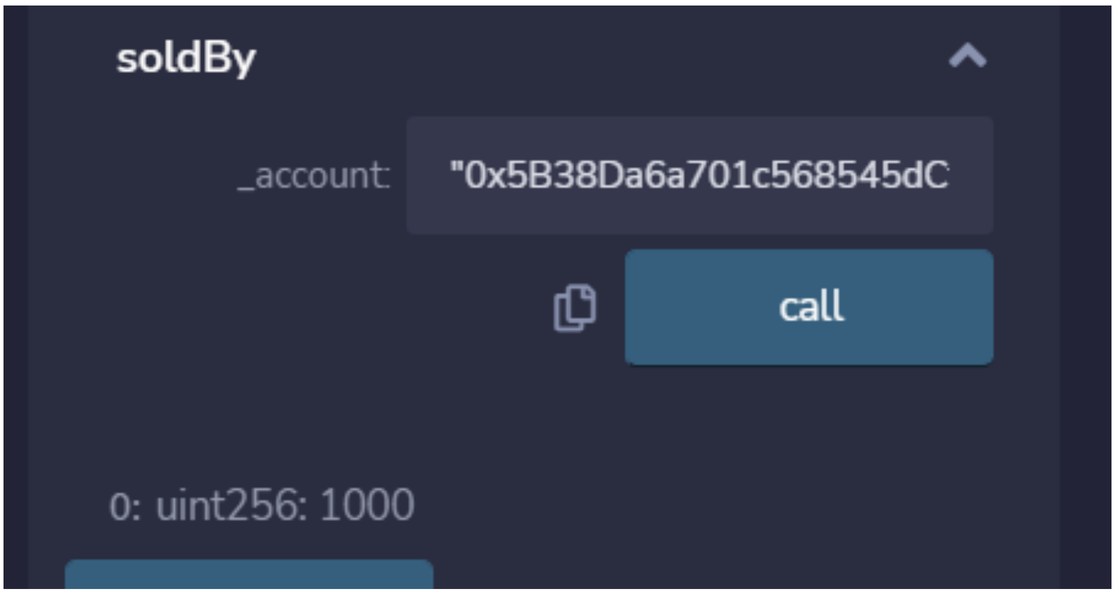
_limit: "1000"

_nft: "0xd9145CCE52D386f25491

_opening: true

2. Mint token exceeded the TOTAL_SUPPLY value

3.result:



```
"0x5B38Da6a701c568545dC
0: uint256: 1000
```

Recommendations

In function *setMaxAmount*, apply code to check the parameter _amount not bigger than TOTAL_SUPPLY. In function *setMaxLimit*, apply code to check the _limit not exceeding the value of max.

Following provides the enhanced code:

```
function setMaxAmount(uint32 _amount)
    external
    onlyOwner
    onlyPositive(_amount)
{

   require(_amount<TOTAL_SUPPLY, "new amount shoud bigger than before")
    max = _amount;
    emit SettedMaxAmount(max);
}


function setMaxLimit(uint32 _limit)
        external
        onlyOwner
        onlyPositive(_limit)
    {
       require(_limit<max, "new amount shoud bigger than before")
        limit = _limit;
        emit SettedMaxLimit(_limit);
    }
```

3.2 Improper Setting of Opening flag

ID: NVE-002	Location: CrowdSale.sol
Severity: High	Category: Business Issue
Likelihood: Low	
Impact: Medium	

Description

setOpening is used to set the start of presale status, *setClosing* is used to set the start of the public sale, but the opening flag is not being set to false after calling *setClosing* with the parameter set to true, which will allow the whitelist users perform exchange of the token with the presale price after the presale stage.

```
    function setClosing(bool _closing) external onlyOwner {
        closing = _closing;
        emit PublicSaleStarted(closing);
    }
```

Recommendations

When the closing value is **true**, the opening flag should be set to **false**

Recommend fixed code:

```
function setClosing(bool _closing) external onlyOwner {
    closing = _closing;
    if(closing==true)
    {
          opening=false;
    }
    emit PublicSaleStarted(closing);
}
```

3.3 Risks in the authority transfer mechanism

ID: NVE-003	Location: CrowdSale.sol
Severity: High	Category: Business Issues
Likelihood: Meduim	

Impact: High

Description

In the function *setPreSalePrice* and *setPublicSalePrice* , when the functions are triggered, the owner will set the new PreSale price and public sale price.
In the contract CrowdSale.sol , the role *onlyOwner* has authority over the following functions:

- *setPreSalePrice*(): set the new price at the presale
- *setMaxAmount*(): will set the max amount
- *setOpening*(): set the start of presale status
- *setClosing*() : set the end of the presale flag

Compromise of *onlyOwner* account may allow a hacker to take advantage of this authority

Exploitation Scenario:

In the event the private key of the owner was hacked or stolen.

Scenario 1: buy the NFT token with very low cost:

1. Attacker can call *setPreSalePrice()* with a very small the parameter
2. Then buys most of the NFT, which will cost very little ether

Scenario 2: enlarge the NFT total amount and the control the token price in market

Recommendations

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent

Short Term:

Timelock and Multi sign (⅔, ⅗) combination mitigate by delaying the sensitive operation and avoiding a single point of key management failure. Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations; AND Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised; AND A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, mitigate by applying decentralization and transparency. Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations; AND Introduction of a DAO/governance/voting module to increase transparency and user involvement; AND A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered fully resolved.

- Renounce the ownership and never claim back the privileged roles; OR
- Remove the risky functionality.
- Noted: Recommend considering the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources

3.4 grantLimits Function is not compared with the Total Supply of the token

ID: NVE-004	Location:CrowdSale.sol
Severity: Low	Vulnerability Type: Business Issues
Likelihood: Low	
Impact: Low	

Description

The function grantLimits() is used to limit the max grant amount of the accounts, so it is used to compare every account with max value.

However the comparison of the total amounts of token with the total supply is missing. The total grant accounts' token amounts may be bigger than total supply(777). e.g : **for every account limit with the value of 10, and with 78 accounts, the total grant value will be 78*10 ,which will have 780 tokens granted and exceeded the total supply of 777.**

```
function grantLimits(address[] memory _accounts, uint256[] memory _limits)
        external
        onlyOwner
    {
        require(
            _accounts.length == _limits.length,
            "_accounts does not match _limits length"
        );
        for (uint256 index = 0; index < _accounts.length; index++) {
            address account = _accounts[index];
            require(_limits[index] <= max, "Exceeded maximum quantity limit");
            quotas[account] = _limits[index];
            //super.grantRole(MINER_ROLE, account);
        }
    }
```

Recommendation

At code line 3895, add a variable in the contract with the code: *uint256 public totalgrant=0* and initialize the value in function *granLimits* every time with 0, and then calculate the sum of the limitation, after that in the for loop, compare the totalgrant with TOTAL_SUPPLY, in this way *grantLimits* exceeding the total supply will be prevented.

Enhanced Code:

```
    function grantLimits(address[] memory _accounts, uint256[] memory _limits)
        external
        onlyOwner
    {
        require(
            _accounts.length == _limits.length,
            "_accounts does not match _limits length"
        );

       totalgrant=0;
        for (uint256 index = 0; index < _accounts.length; index++) {
            address account = _accounts[index];
            require(_limits[index] <= max, "Exceeded maximum quantity limit");
            quotas[account] = _limits[index];

            totalgrant = totalgrant + _limits[index];
            require(totalgrant < TOTAL_SUPPLY,"invaild grant limit");

        }
    }
```

3.5 giftLimit variable declared but not used

ID: NVE-005	Location:CrowdSale.sol
Severity: Low	Vulnerability Type: Business Issues
Likelihood: Low	
Impact: Low	

Description

The variable *giftLimit* is seemingly used to limit the max amount of gift, but it is not being declared in the contract code logic. The contract function *gift(uint256* *_amount, bytes calldata signature)* should use this value to limit the total gifts amount to prevent exceeding the maximum gifts limitation.

Recommendation

Add a new variable e.g: *uint256 public totalgift=0;* at the line 3896, and in the function gift, calculate the total gift every time the gift function is called, and then compare the total gift value with *giftLimit* to prevent the gifts limit from exceeding the *giftLimit*

Recommend fixed code:

```
    function gift(uint256 _amount, bytes calldata signature)
        external
        requiresGift(signature)
    {
        require(!opening, "Gift time is over");
        require(_amount <= limit, "More than one purchase");
        address miner = msg.sender;
        free[miner] = _amount.add(free[miner]);
        require(free[miner] <= max, "Exceeded maximum quantity limit");
        totalgift=totalgift+_amount;
        require(totalgift <= giftLimt,"Exceeded the maximum gifts limit" );
        token.mint(miner, _amount);
    }
```

3.6 Multiple Compiler Versions Declared

ID: NVE-006
Severity: informational
Likelihood: informational
Impact: informational

Location:
CrowdSale.sol, NFTERC721A.sol
Vulnerability Type: Compile Issues

Description

The latest version of pragma solidity ^0.8.7 has been released on NFTERC721A.sol and crowdsales.sol. Multiple solidity versions were used in the compiler components. Inconsistent solidity versions can potentially lead to less secured solidity code and less efficient gas optimization capabilities.

NFTERC721A.sol

e.g：`391: pragma solidity ^0.8.0;`

```
      787: pragma solidity ^0.8.4;
```

Recommendations

Remove redundant solidty compiler version declaration, keep only ***pragma solidity ^0.8.7***

3.7 Improper Usage Of Public Functions

ID: NVE-007
Severity: Informational
Likelihood: Informational
Impact: Informational

Location:CrowdSale.sol,NFTERC721A.sol
Category: Security Features

Description

'Public' functions that not being called in the contract should be declared as External to save gas.

The below functions need to change from public to external:

CorwdSale.sol:

```

    function freeMinted(address _account) public view returns (uint256) {
        return free[_account];
    }

    function allowance(address _account) public view returns (uint256) {
        uint256 result;
        if (closing) {
            (, result) = max.trySub(sold[_account]);
            return result;
        }
        (, result) = quotas[_account].trySub(sold[_account]);
        return result;
    }

    function soldBy(address _account) public view returns (uint256) {
        return sold[_account];
    }
```

NFTERC721A.sol:

```
  function pause() public virtual {
        require(
            hasRole(PAUSER_ROLE, _msgSender()),
            "NFT: must have pauser role to pause"
        );
        _pause();
    }

    /**
     * @dev Unpauses all token transfers.
     *
     * See {ERC721Pausable} and {Pausable-_unpause}.
     *
     * Requirements:
     *
     * - the caller must have the `PAUSER_ROLE`.
     */
    function unpause() public virtual {
        require(
            hasRole(PAUSER_ROLE, _msgSender()),
            "NFT: must have pauser role to unpause"
        );
        _unpause();
    }

    function current() public view returns (uint256) {
        return _totalMinted();
    }

    function contractURI() public view returns (string memory) {
        return collectionURI;
    }

    function setContractURI(string memory _contractURI) public onlyOwner {
        collectionURI = _contractURI;
    }

    /// @dev Sets the base token URI prefix.
    function setBaseTokenURI(string memory _baseTokenURI) public onlyOwner {
        baseTokenURI = _baseTokenURI;
    }
```

Recommendations

Consider using the external attribute for functions not being called from the contract

CorwdSale.sol:

```
    function freeMinted(address _account) external view returns (uint256) {
        return free[_account];
    }

    function allowance(address _account) external view returns (uint256) {
        uint256 result;
        if (closing) {
            (, result) = max.trySub(sold[_account]);
            return result;
        }
        (, result) = quotas[_account].trySub(sold[_account]);
        return result;
    }

    function soldBy(address _account) external view returns (uint256) {
        return sold[_account];
    }
```

NFTERC721A.sol

```

  function pause() external virtual {
        require(
            hasRole(PAUSER_ROLE, _msgSender()),
            "NFT: must have pauser role to pause"
        );
        _pause();
    }

    /**
     * @dev Unpauses all token transfers.
     *
     * See {ERC721Pausable} and {Pausable-_unpause}.
     *
     * Requirements:
     *
     * - the caller must have the `PAUSER_ROLE`.
     */
    function unpause() external virtual {
        require(
            hasRole(PAUSER_ROLE, _msgSender()),
            "NFT: must have pauser role to unpause"
        );
        _unpause();
    }

    function current() external view returns (uint256) {
        return _totalMinted();
    }

    function contractURI() external view returns (string memory) {
        return collectionURI;
    }

    function setContractURI(string memory _contractURI) external onlyOwner {
        collectionURI = _contractURI;
    }

    /// @dev Sets the base token URI prefix.
    function setBaseTokenURI(string memory _baseTokenURI) external onlyOwner {
        baseTokenURI = _baseTokenURI;
    }
```

3.8 Missing Emit Events

ID: NVE-008
Severity: Informational
Likelihood: Informational
Impact: Informational

Location:CrowdSale.sol
Vulnerability Type: Business Issues

Description

The function *pauseClaimablePeriod()* affects the sensitive status of the contract and should emit events as notifications to users.

```
    function setMaxAmount(uint32 _amount)
        external
        onlyOwner
        onlyPositive(_amount)
    {
        max = _amount;
    }

    function setMaxLimit(uint32 _limit)
        external
        onlyOwner
        onlyPositive(_limit)
    {
        limit = _limit;
    }
```

Recommendations

Consider adding an event for the sensitive actions, and emit it in the function in CrowdSale.sol

```
function setMaxAmount(uint32 _amount)
    external
    onlyOwner
    onlyPositive(_amount)
{

   require(_amount<TOTAL_SUPPLY, "new amount shoud not bigger than TOTAL_SUPPLY")
    max = _amount;
    emit SettedMaxAmount(max);
}

function setMaxLimit(uint32 _limit)
        external
        onlyOwner
        onlyPositive(_limit)
    {
       require(_limit<max, "limit should not bigger than max")
        limit = _limit;
        emit SettedMaxLimit(_limit);
    }
```

3.9 Optimization of Variable Type

ID: NVE-009
Severity: Informational
Likelihood: Informational
Impact: Informational

Location: CrowdSale.sol
Vulnerability Type: Business Issues

Constant state variables should be declared as constant to save gas.

```
    uint256 public  TOTAL_SUPPLY = 777;
```

Recommendations:

Set TOTAL_SUPPLY variable to const as shown in the following code snippet:

```
uint256 public constant TOTAL_SUPPLY = 777;
```

4 Conclusion

In this audit, we thoroughly analyzed NFTSTART smart contract implementation. The findings outlined in section 1.4 require corrective actions and attention. To improve this report, we greatly appreciate any constructive feedback or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5 Appendix

5.1 Basic Coding Assessment

5.1.1 Apply Verification Control

- Description: The security of apply verification
- Result: Not found
- Severity: Critical

5.1.2 Authorization Access Control

- Description: Permission checks for external integral functions

• Result: Not found
• Severity: Critical

5.1.3 Forged Transfer Vulnerability

• Description: Assess whether there is a forged transfer notification vulnerability in the contract
• Result: Not found
• Severity: Critical

5.1.4 Transaction Rollback Attack

• Description: Assess whether there is transaction rollback attack vulnerability in the contract.
• Result: Not found
• Severity: Critical

5.1.5 Transaction Block Stuffing Attack

• Description: Assess whether there is transaction blocking attack vulnerability.
• Result: Not found
• Severity: Critical

5.1.6 soft_fail Attack Assessment

• Description: Assess whether there is soft_fail attack vulnerability.
• Result: Not found
• Severity: Critical

5.1.7 hard_fail Attack Assessment

• Description: Examine for hard_fail attack vulnerability
• Result: Not found
• Severity: Critical

5.1.8 Abnormal Memo Assessment

• Description: Assess whether there is abnormal memo vulnerability in the contract.
• Result: Not found
• Severity: Critical

5.1.9 Abnormal Resource Consumption

• Description: Examine whether abnormal resource consumption in contract processing.

- Result: Not found
- Severity: Critical

5.1.10 Random Number Security

- Description: Examine whether the code uses insecure random number.
- Result: Not found
- Severity: Critical

5.2 Advanced Code Scrutiny

5.2.1 Cryptography Security

- Description: Examine for weakness in cryptograph implementation.
- Results: Not Found
- Severity: High

5.2.2 Account Permission Control

- Description: Examine permission control issue in the contract
- Results: Not Found
- Severity: Medium

5.2.3 Malicious Code Behaviour

- Description: Examine whether sensitive behaviour present in the code
- Results: Not found
- Severity: Medium

5.2.4 Sensitive Information Disclosure

- Description: Examine whether sensitive information disclosure issue present in the code.
- Result: Not found
- Severity: Medium

5.2.5 System API

- Description: Examine whether system API application issue present in the code
- Results: Not found
- Severity: Low

References

[1] MITRE. CWE-191: Integer Underflow (Wrap or Wraparound). https://cwe.mitre.org/data/ definitions/191.html.

[2] MITRE. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197. html.

[3] MITRE. CWE-400: Uncontrolled Resource Consumption. https://cwe.mitre.org/data/ definitions/400.html.

[4] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440. html.

[5] MITRE. CWE-684: Protection Mechanism Failure. https://cwe.mitre.org/data/definitions/ 693.html.

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

[7] MITRE. CWE CATEGORY: Behavioral Problems.
https://cwe.mitre.org/data/definitions/438. html.

[8] MITRE. CWE CATEGORY: Numeric Errors.
https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE CATEGORY: Resource Management Errors.
https://cwe.mitre.org/data/ definitions/399.html.

[10] OWASP. Risk Rating Methodology.
https://www.owasp.org/index.php/OWASP_Risk_ Rating_Methodology.