# Introduction

A smart contract security audit review of the noramp protocol ERC721N was done by sparkware's auditor, 0xladboy233, with a focus on the security aspects of the application's implementation.

# About Auditing Firm and Brand

Octane Security is an AI-driven blockchain security firm dedicated to safeguarding digital assets and transactions with cutting-edge technology.

https://www.octane.security/

Octane act as middleman to faciliate the audit.

Sparkware security offers cutting edge and affordable smart contract auditing solution. The auditors get reputatoin and skill by consistently get top place in bug bounty and audit competition.

Our past audit prject including optimism and notional finance and graph protocol. Below is a list of comprehensive security review and research: https://github.com/JeffCX/Sparkware-audit-portfolio

# Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

# Security Assessment Summary

***review commit hash* - [7e169014ba07696c8196c2a6b60270a0054e645c]**

## Scope

The following smart contracts were in scope of the audit:

- `ERC721N.sol`

## Findings

| ID | Title | Severity |
|---|---|---|
| M-1 | ERC721N is not compatible with ERC20 token that does not return boolean | Medium |
| L-1 | Lack of function to claim or remove the token | Low |
| L-2 | depositReserves function can be optimized | Low |
| L-3 | Owner of the contract can remove funds anytime | Low |
| L-4 | Approved owner cannot redeem reserves | Low |
| L-5 | SafeMint should use _safemint instead of the current implementation | Low |

This table is organized to show the ID, title, and severity of each finding, formatted to be similar to the original table you provided.

# M - ERC721N is not compatible with ERC20 token that does not return boolean when calling transfer / transferFrom

## Description

In the current codebase,

the code enforce the return value of external transfer and transferFrom

For example, the code for depositReserves is:

```
bool success = reserveTokenAddress.transferFrom(
    msg.sender,
    address(this),
    _amount
);
if (!success) {
    revert ERC20TransferFailed(msg.sender, _amount);
}
```

and the code for redeemReserves is

```
// Attempt to transfer the ERC20 tokens to the caller
bool success = reserveTokenAddress.transfer(msg.sender, amount);
if (!success) {
    revert ERC20TransferFailed(msg.sender, amount);
}
```

https://github.com/d-xo/weird-erc20?tab=readme-ov-file#missing-return-values

> Some tokens do not return a bool (e.g. USDT, BNB, OMG) on ERC20 methods. see here for a comprehensive (if somewhat outdated) list.

> Some tokens (e.g. BNB) may return a bool for some methods, but fail to do so for others. This resulted in stuck BNB tokens in Uniswap v1 (details).

then enforcing a return value for token transfer will revert the transaction

### Recommendation

use openzeppelin safeTransfer

https://docs.openzeppelin.com/contracts/5.x/api/token/erc20#SafeERC20

## L - Lack of function to claim or remove the token

### Description

In the codebase, anyone can deposit reserves and nft holder can burn nft in exchange for token,

However, there is lack of function to remove the token from the contract other than redeemReserves

Consider the case 1:

1. user A deposit 10000 token to contract.
2. owner mint 2 nft, nft token id 1 can claim 3000 tokens, nft token id 2 can claim 7000 tokens
3. nft token id 1 is sold to user alice, but alice lose his wallet access
4. the 3000 token for alice to claim is forever locked in the contract.

Consider case 2:

Certain token has rebasing behavior

some of the well known one are

https://blog.lido.fi/steth-the-mechanics-of-steth/

and blast WETH and USDB token

https://docs.blast.io/building/guides/weth-yield

However, because there is lack of function to remove the token from the contract,

the additionally rebased yield accured to the contract, not users, which leads to lose of fund.

assume the contract hold 1 million token and the token has 4% positive rebasing rate yearly, the 4% of 1 million is lost and locked

## Recommendation

add a function for owner to claim the additional token.

```
function claimExcessiveToken() public {
    require(msg.sender == owner, "only owner can claim fund");
    uint256 balance = reserveTokenAddress.balanceOf(this);
    if (balance > unclaimedReserveBalance) {
        reserveTokenAddress.safeTransfer(msg.sender, balance -
unclaimedReserveBalance);
    }
}
```

# L – depositReserves function can be optimized

## Description

In depositReserves function,

```
    modifier checkAllowance(uint amount) {
        if (reserveTokenAddress.allowance(msg.sender, address(this)) <
amount) {
            revert AllowanceTooLow(amount);
        }
        _;
    }

    /**
     * @dev Allows users to deposit ERC20 tokens into the contract.
Requires prior approval.
     * @param _amount The amount of ERC20 tokens to deposit.
     */
    function depositReserves(
        uint256 _amount
    ) external checkAllowance(_amount) nonReentrant {
        bool success = reserveTokenAddress.transferFrom(
            msg.sender,
            address(this),
            _amount
        );
        if (!success) {
            revert ERC20TransferFailed(msg.sender, _amount);
        }
        emit DepositReserves(msg.sender, _amount);
    }
```

first, to save gas, the modifier checkAllowance check can be removed,

if there is insufficient allowance, the transferFrom will revert anyway,

second, to save gas, the function depositReserves can be reserved,

because the reserves balance uses balanceOf to track balance

```
 uint reserveBalance = reserveTokenAddress.balanceOf(address(this));
if (reserveBalance < unclaimedReserveBalance + _quantity) {
    revert InsufficientReserveBalance(_quantity, reserveBalance);
}
uint256 tokenId = _nextTokenId++;
```

then anyone can just transfer the token to the contract directly instead of calling depositReserves

## Recommendation

the function depositReserves can be removed, or if the protocol wants another way to track the balance, the code can use internal accounting to track the state reserveBalance

```
    function depositReserves(
        uint256 _amount
    ) external checkAllowance(_amount) nonReentrant {
        uint256 balanceBefore = reserveTokenAddress.balanecOf(this);
        reserveTokenAddress.safeTransferFrom(
            msg.sender,
            address(this),
            _amount
        );
        uint256 afterBefore = reserveTokenAddress.balanecOf(this);
        uint256 amount = afterBefore - balanceBefore;
```

```
        reserveBalance += amount
        emit DepositReserves(msg.sender, amount);
    }
```

# L – Owner of the contract can remove fund or steal fund from the contract anytime

## Description

the current code gives the owner very high privilege:

```
    function safeMint(address _to, uint256 _quantity) internal nonReentrant {
        // Only the owner of the contract can mint new tokens
        if (msg.sender != _owner) {
            revert Unauthorized(msg.sender);
        }
        // User must select a valid amount of ERC20 tokens to reserve
        if (_quantity <= 0) {
            revert AmountMustBeGreaterThan0(_quantity);
        }
        uint reserveBalance = reserveTokenAddress.balanceOf(address(this));
        if (reserveBalance < unclaimedReserveBalance + _quantity) {
            revert InsufficientReserveBalance(_quantity, reserveBalance);
        }
        uint256 tokenId = _nextTokenId++;
        unclaimedReserveBalance += _quantity;
        tokenERC20Balances[tokenId] = _quantity;
        _mint(_to, tokenId);
        emit ERC721NMinted(_to, tokenId, _quantity);
    }
```

assume the contract is deployed,

user alice is the owner,

user bob deposit 1000 token as reserve balance.

uesr alice at any time can always mint an additional nft and claim the 1000 token burn burning the nft because owner can set _quantity to 1000 token.

### Recommendation

This is a centralization risk, given anyone can deploy token, such risk still worth highlighting even there is current no good fix.

## L – Approved owner cannot redeem reserves

### Description

when calling redeemReserves

```
    function redeemReserves(uint256 tokenId) external nonReentrant {
        uint256 amount = tokenERC20Balances[tokenId];

        if (amount == 0) {
            revert NoERC20BalanceToRedeem(tokenId);
        }
        if (msg.sender != ownerOf(tokenId)) {
            revert NotOwnerOfToken(msg.sender, tokenId);
        }
```

only the owner of the nft can redeem reserves,

even the owner approves an operator, the operator cannot redeem the reserves,

### Recommendation

```
    function redeemReserves(uint256 tokenId) external nonReentrant {
          uint256 amount = tokenERC20Balances[tokenId];

          if (amount == 0) {
              revert NoERC20BalanceToRedeem(tokenId);
          }
          address owner = ownerOf(tokenId);
          if (msg.sender != owner && !isApprovedForAll(owner, msg.sender) &&
  !_getApproved(tokenId) != msg.sender ) {
              revert NotOwnerOfToken(msg.sender, tokenId);
          }
```

## L – SafeMint should use _safemint

### Description

In the current code, the _mint function is called inside safeMint

https://docs.openzeppelin.com/contracts/2.x/api/token/erc721#ERC721-_safeMint-address-uint256-

Avoid loss of nft when the code mint the nft to a smart contract that is not capable of handling nft transfer,

_safeMint should be used in function safeMint to stay ERC721 compliant. safeMint ensure that if the receiver of nft is a smart contract,

the smart contract must implement onERC721Received hook.

## Recommendation

use _safeMint instead of _mint inside ERC721N.so#safeMint