verichains

SECURITY AUDIT OF

SHOPNEXT SMART CONTRACT

Public Report

Nov 30, 2022

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Nov 30, 2022. We would like to thank the ShopNext for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the ShopNext Smart Contract. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contract code, along with some recommendations. ShopNext team has resolved and updated most of the issues following our recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About ShopNext Smart Contract

ShopNEXT is a Web3 Loyalty Platform powered by VISA and BNB chain. We utilize card payment, NFT and gamification to invent the Shop-To-Earn model.ShopNEXT allows users to earn token rewards from daily shopping while helping merchants grow their businesses. ShopNEXT token is a kind of loyalty point but the difference is that it is developed on blockchain and is exchangeable.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the ShopNext Smart Contract. It was conducted on commit `37a98206d4240aa7c1d8e936b185e78c86aafed0` from git repository link: *https://github.com/Shopnext-io/shopnext-sc-audit*.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
ff89f230790623dd827391cf0460e4967352448ef894b6ed7605e8744558be3e	SNSwap.sol
e3861f052b969b34e7178b17a99b4ed97a899d8c867d77bafcd80ff18a63f91e	Next.sol
91bd883b6b526a35793ca925b20cb429db7f6d32262d0d203b4c9dc209452513	NextMooner.sol
7e7d395ed443a79b92235c008b4a51a8cadf495c1efb665977732927e388d134	MarketPlace.sol
da6efc41ac269c3ee96e6c0fefedc85a96e74285aa1ef617fc72372386933360a	NFTCard.sol
e043209ab3bc885de8b6879cdffc8e40d6b1a9d708896922e20d39fad57db937	libraries/EnumerableMap.sol
e0ba810c1efe70dd189391f12a8bc3bb88158394adb5e094c36aed365cb4444d	Withdraw.sol
83708cc1a7a38c2452510e7108c3b898a7936a867da18fb51046e790895f8c9b	STE.sol

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The ShopNext Smart Contract was written in `Solidity` language, with the required version to be `^0.8.4` and `^0.8.2`. The source code was written based on OpenZeppelin's library.

2.1.1. NEXT Token contract

NEXT Token contract only extends the `ERC20` contract. At deployment, 100,000,000 ($x10^{18}$) are minted for contract deployer.

2.1.2. STE Token contract

NEXT Token contract extends the `ERC20`, `ERC20Burnable` and `Ownable` contracts. With `Ownable`, by default, Token Owner is contract deployer, but he can transfer ownership to another address at any time. `ERC20Burnable` allows token holders to destroy both their own tokens and those that they have an allowance for.

At deployment, 1,000,000 ($x10^{18}$) are minted for contract deployer, and he can also mint to a specified address at any time.

2.1.3. NextMarket contract

This is the marketplace contract in the Next Smart Contracts, which extends `ReentrancyGuard` and `Ownable` contract. With `Ownable`, by default, Contract Owner is the contract deployer, but he can transfer ownership to another address at any time.

The contract includes the following main functions:

- listForSale: allows the seller to list NFT for sale
- delist: allows seller to cancel NFT listing
- changePrice: allows seller to change price NFT listed
- buy: allows the buyer to buy NFT listing
- setSNAddress, setReceiveFunAddress, setMarketFeeInBps, setNFTCardAddress: allows admin config NEXT token address, address receive fund from market, market fee, address NFT Card
- offer: allows the buyer to offer a specific price for NFT that has been listed or not. The buyer can change the offer price
- takeOffer: the seller accepts the price and transfers NFT to buyer
- cancelOffer: allows the buyer to cancel the offer

2.1.4. NextMooner contract

This contract manages the list mooner account, and allows account up- and down- moon. The contract extends `Ownable` contract, by default, Contract Owner is the contract deployer, but the owner can transfer ownership to another address at any time.

The contract includes the following main functions:

- upgradeMoon: allows account upgrade to mooner. Users will have to pay amount NEXT token, it is like staking. When user end mooner, they can claim token + reward (config by apr index)
- downgradeMoon: allows account request downgrade mooner
- cancelDowngradeMoon: allows account cancel request downgrade mooner
- claimToken: allows user claim token + reward
- setApr, setExpiredTime, setNextToken, addSigner,...: allow admin config contract.

Note: The mooner life cycle is as follows:

- Time mooner fix in a $durationTime ($durationTime config by admin). After $durationTime if account does not downgrade mooner auto review - If user request downgrade moon then user has to wait util end the current $durationTime

2.1.5. NFTCard contract

This contract is ERC721 NFT Card contract, which extends `ERC721`, `ERC721Enumerable`, `Ownable`, by default, Contract Owner is the contract deployer, but he can transfer ownership to another address at any time. `ERC721Enumerable` is used to track the owners of an NFT on-chain.

The contract includes the following main function:

- claimNft: allows user claim NFT when getting signature from admin.

2.1.6. SNSwap contract:

This contract support account swap from Next token from version 1 to version 2 according to the ratio configurable by the admin which extends `Ownable`, by default, Contract Owner is the contract deployer, but he can transfer ownership to another address at any time.

Note: whiteList users will be transferred immediately instead of waiting like other normal users.

2.1.7. Withdraw contract:

This contract support user withdrawing NEXT, STE token from the system when getting a signature from admin. It extends `Ownable` contract, by default, Contract Owner is the contract deployer, but the owner can transfer ownership to another address at any time.

The contract has 2 withdrawal rules:

- Each user only withdraws tokens for a period of time configured by the admin. (ex: user only withdraws token once a day)
- Total token withdrawal of system has a quota amount.

The contract includes the following main functions:

- claim: allows user withdraw token when getting signature from admin
- setMaximumPerUint, setTimeUint, setLockTimeWithdrawPerUser, setAllowToken, setstoreTokenAddress, add/remove Signer.

2.2. Findings

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

2.2.1. Marketplace.sol - Front running in `takeOffer` function CRITICAL

Attackers can list an item by `offer` with high price and wait for a seller takes offer it with `takeOffer`. While waiting, attackers listen for pending transactions and when a seller takes offer that item, they change the offer price by making an `offer` with a higher gas price (higher gas price transaction is usually mined first) than the `takeOffer` transaction to reoffer the item at a lower price, and as a result, the seller sells the item for less money than he sees in the market.

For example: An attacker offers an NFT item with 1000 tokens in the marketplace, the seller finds the price reasonable and accepts this offer to sell it with 1000 tokens. The attacker listens to pending transactions and knows that someone is selling the item for 1000 tokens and send `offer` transactions to reoffer the item for 1 token with higher gas price and get mined before the `takeOffer` transaction. The result is user lost 999 SN tokens to attacker for that item.

```
function takeOffer(uint256 tokenId, address _buyer)
        external
        onlyNftOwner(tokenId)
        nonReentrant
    {
        //get price buyer offer
        (, uint256 offeredValue) = nftOnOffers[tokenId].tryGet(_buyer);
        address seller = msg.sender;

        //validate data
        require(offeredValue > 0, "SN: no offer found");
        require(_buyer != seller, "SN: cannot buy your own nft");

        //remove offer order
        nftOnOffers[tokenId].remove(_buyer);
        nftOnOffersOfUser[_buyer].remove(tokenId);
```

```
        _makeTransaction(tokenId, _buyer, seller, offeredValue,1);

    }
```

RECOMMENDATION

Adding `price` parameter to `takeOffer` function to revert when NFT price is lower than the price the seller want to sell.

```
function takeOffer(uint256 tokenId, address _buyer, uint256 price)
        external
        onlyNftOwner(tokenId)
        nonReentrant
    {
        //get price buyer offer
        (, uint256 offeredValue) = nftOnOffers[tokenId].tryGet(_buyer);
        address seller = msg.sender;

        //validate data
        require(price == offeredValue, "SN: offer price invalid");
        require(offeredValue > 0, "SN: no offer found");
        require(_buyer != seller, "SN: cannot buy your own nft");

        //remove offer order
        nftOnOffers[tokenId].remove(_buyer);
        nftOnOffersOfUser[_buyer].remove(tokenId);

        _makeTransaction(tokenId, _buyer, seller, offeredValue,1);
    }
```

UPDATES

- *Nov 30, 2022*: This issue has been acknowledged and fixed by the ShopNext team.

2.2.2. Marketplace.sol - The attacker can buy the item for 0 token after the item has just been approved CRITICAL

When users approve item, attacker can see the approved transaction for that item and can buy this item for 0 by calling `buy` function with token ID and 0.

Progress:

- Step 1: the user approves the market for any item (ex. approve NFT item with id: 123).
- Step 2: the attacker can see the approved transaction for that item.
- Step 3: the attacker can buy that item at 0 price by calling the function: `buy(123, 0)` before the item is listed and after it is removed from the market.

```
function buy(uint256 tokenId, uint256 price) external nonReentrant {
        //get price of tokenId
        uint256 priceForList = nftOnSales[tokenId];

        require(price == priceForList, "SN: amount invalid");
        //get address of seller
        address seller = nftCardAddress.ownerOf(tokenId);
        address buyer = msg.sender;

        require(buyer != seller, "SN: cannot buy your nft");

        //set price listing of tokenId
        nftOnSales[tokenId] = 0;
        _makeTransaction(tokenId, buyer, seller, price,0);

    }
```

RECOMMENDATION

Should require `priceForList` greater than 0.

```
function buy(uint256 tokenId, uint256 price) external nonReentrant {
        //get price of tokenId
        uint256 priceForList = nftOnSales[tokenId];

        require(priceForList > 0, "SN: not listed");
        require(price == priceForList, "SN: amount invalid");
        //get address of seller
        address seller = nftCardAddress.ownerOf(tokenId);
        address buyer = msg.sender;

        require(buyer != seller, "SN: cannot buy your nft");

        //set price listing of tokenId
        nftOnSales[tokenId] = 0;
        _makeTransaction(tokenId, buyer, seller, price,0);
    }
```

UPDATES

- *Nov 30, 2022*: This issue has been acknowledged and fixed by the ShopNext team.

2.2.3. Marketplace.sol - The user may buy wrong item and wrong token payment if the owner change contract address CRITICAL

The user can buy a item in old contract address and recieve a NFT in new contract address if the item is still listed and offered on the market before the NFT contract address is changed, same as token payment contract address.

RECOMMENDATION

There are 2 ways to fix it:

- Check that there are no more items on sale or offer before changing the NFT contract address.
- Add a mapping between tokenId and NFT contract address and Token paypent contract address.

UPDATES

- *Nov 30, 2022*: This issue has been acknowledged by the ShopNext team.

2.2.4. SNSwap.sol - Whitelist users can get more tokens than they swap MEDIUM

When swapping `whiteList` users will receive `newSNToken` immediately instead of waiting through each `block.number` like normal users. But the system then saves the swap history and they can get more tokens when they use the `claim()` function.

```
function swap(uint256 amount) external{
        oldSNToken.safeTransferFrom(_msgSender(),deadAddress, amount);

        uint256 released =0;
         if(whiteList[msg.sender]){
            released = amount * rate/BPS;
            newSNToken.safeTransfer(msg.sender,released);
         }
        listSwap.push(Swap(block.number,amount,unlockDuration,released));
        listSwapOfUsers[msg.sender].push(listSwap.length -1);
        emit SwapToken(msg.sender, amount, block.number, unlockDuration);
    }
    function claim(address user) external{
        uint256[] memory listSwapOfUser = listSwapOfUsers[user];
        uint256 totalClaim ;
        for(uint256 i =0;i< listSwapOfUser.length;i++){
            Swap storage swapItem = listSwap[listSwapOfUser[i]];
            uint256 released = getReleaseAble(listSwapOfUser[i]);
            totalClaim += (released - swapItem.released);
            swapItem.released = released;
        }
        newSNToken.safeTransfer(user, totalClaim* rate/BPS);
    }
```

RECOMMENDATION

Use `if - else` instead of `if` in `swap()` function

```
function swap(uint256 amount) external{
        oldSNToken.safeTransferFrom(_msgSender(),deadAddress, amount);

        uint256 released =0;
        if(whiteList[msg.sender]){
            released = amount * rate/BPS;
            newSNToken.safeTransfer(msg.sender,released);
        } else {
            listSwap.push(Swap(block.number,amount,unlockDuration,released));
            listSwapOfUsers[msg.sender].push(listSwap.length -1);
        }
        emit SwapToken(msg.sender, amount, block.number, unlockDuration);
    }
```

UPDATES

- *Nov 30, 2022*: This issue has been acknowledged and fixed by the ShopNext team.

2.2.5. Marketplace.sol - The offer price should be greater than 0 LOW

Since the require of the current price of the item in the `takeOffer` function is greater than 0, if the offer price is set to 0, the offer will be useless.

```
function offer(uint256 _tokenId, uint256 _price) external nonReentrant {
        require(_price >= 0, "SN: price invalid");
        address buyer = msg.sender;

        //get current price offer of same buyer and heroID
        (, uint256 currentOffer) = nftOnOffers[_tokenId].tryGet(buyer);

        ...
    }
```

RECOMMENDATION

Use `>` instead of `>=`

```
function offer(uint256 _tokenId, uint256 _price) external nonReentrant {
        require(_price > 0, "SN: price invalid");
        address buyer = msg.sender;

        //get current price offer of same buyer and heroID
        (, uint256 currentOffer) = nftOnOffers[_tokenId].tryGet(buyer);

        ...
    }
```

UPDATES

- *Nov 30, 2022*: This issue has been acknowledged and fixed by the ShopNext team.

2.2.6. NextMooner.sol - `cancelDowngradeMoon()` function does not implement the correct logic LOW

`cancelDowngradeMoon()` function only change the variable `startDownTime` that is not used in the contract, without changing the necessary variable `endTimeVip`.

```
function cancelDowngradeMoon() external {
        require(isMoon(_msgSender()), "SN: user is not mooner");
        Mooner storage mooner = moonerList[_msgSender()];
        require(mooner.endTimeVip >0 && mooner.endTimeVip > block.timestamp,"SN:user not
request down moon" );
        mooner.startDownTime = 0;
        emit ChangeMoonerStatus(
            _msgSender(),
            2,
            block.timestamp,
            0,
            mooner.amountLock
        );
    }
```

RECOMMENDATION

Change `endTimeVip` variable to 0 to continue moon.

```
function cancelDowngradeMoon() external {
        require(isMoon(_msgSender()), "SN: user is not mooner");
        Mooner storage mooner = moonerList[_msgSender()];
        require(mooner.endTimeVip >0 && mooner.endTimeVip > block.timestamp,"SN:user not
request down moon" );
        mooner.startDownTime = 0;
        mooner.endTimeVip = 0;
        emit ChangeMoonerStatus(
            _msgSender(),
            2,
            block.timestamp,
            0,
            mooner.amountLock
        );
    }
```

UPDATES

- *Nov 30, 2022*: This issue has been acknowledged and fixed by the ShopNext team.

2.3. Additional notes and recommendations

2.3.1. Marketplace.sol - Should check if `_tokenId` exists or not before adding offer INFORMATIVE

User can offer any item that doesn't exist.

RECOMMENDATION

Add a require, check if the `_tokenId` is owned by anyone or not.

```
function offer(uint256 _tokenId, uint256 _price) external nonReentrant {
        require(_price > 0, "SN: price invalid");
        // Add check tokenId
        require (IERC721(nftCardAddress).ownerOf(_tokenId) != address(0))

        address buyer = msg.sender;

        //get current price offer of same buyer and heroID
        (, uint256 currentOffer) = nftOnOffers[_tokenId].tryGet(buyer);

        ...
    }
```

UPDATES

- *Nov 30, 2022*: This issue has been acknowledged and fixed by the ShopNext team.

2.3.2. SNSwap.sol - Should require amount of swap greater than 0 to avoid wasting gas INFORMATIVE

With 0 token transfer in `swap` function, user can call it multiple times and increase system memory unnecessarily.

```
function swap(uint256 amount) external{
        oldSNToken.safeTransferFrom(_msgSender(),deadAddress, amount);

        uint256 released =0;
         if(whiteList[msg.sender]){
            released = amount * rate/BPS;
            newSNToken.safeTransfer(msg.sender,released);
         }
        listSwap.push(Swap(block.number,amount,unlockDuration,released));
        listSwapOfUsers[msg.sender].push(listSwap.length -1);
        emit SwapToken(msg.sender, amount, block.number, unlockDuration);
    }
```

RECOMMENDATION

Require the amount to be greater than 0.

```
function swap(uint256 amount) external{
        require(amount > 0, "SN: amount invalid");
        oldSNToken.safeTransferFrom(_msgSender(),deadAddress, amount);

        uint256 released =0;
        if(whiteList[msg.sender]){
            released = amount * rate/BPS;
            newSNToken.safeTransfer(msg.sender,released);
        } else {
            listSwap.push(Swap(block.number,amount,unlockDuration,released));
            listSwapOfUsers[msg.sender].push(listSwap.length -1);
        }
        emit SwapToken(msg.sender, amount, block.number, unlockDuration);
    }
```

UPDATES

- *Nov 30, 2022*: This issue has been acknowledged and fixed by the ShopNext team.

2.3.3. SNSwap.sol - Unused `swapId` variable, `Counter` library and `ClaimToken` event INFORMATIVE

RECOMMENDATION

- Remove unused swapId variable, Counter library
- Add `claim` event to `claim()` function

```
function claim(address user) external{
        uint256[] memory listSwapOfUser = listSwapOfUsers[user];
        uint256 totalClaim ;
        for(uint256 i =0;i< listSwapOfUser.length;i++){
            Swap storage swapItem = listSwap[listSwapOfUser[i]];
            uint256 released = getReleaseAble(listSwapOfUser[i]);
            totalClaim += (released - swapItem.released);
            swapItem.released = released;
        }
        newSNToken.safeTransfer(user, totalClaim* rate/BPS);
        emit ClaimToken(user, totalClaim* rate/BPS);
    }
```

UPDATES

- *Nov 30, 2022*: This issue has been acknowledged and fixed by the ShopNext team.

2.3.4. NextMooner.sol - Typo in `amountInterst` INFORMATIVE

There are some typo in `amountInterst`, the correct should be `amountInterest`.

```
function claimToken() external {
        require(!isMoon(_msgSender()), "SN: user is  mooner");
        Mooner storage mooner = moonerList[_msgSender()];
        uint256 amountClaim;
        uint256 amountInterst;
        amountClaim = mooner.amountLock;
        if (isStakeMode) {
            amountInterst = ((mooner.amountLock *
                apr *
                (mooner.endTimeVip - mooner.startUpTime)) / (365 days * BPS));
        }

        mooner.amountLock = 0;

        if (amountClaim > 0) {
            nextToken.safeTransfer(_msgSender(), amountClaim);
            emit Claimed(msg.sender, amountClaim);
        }
        if (amountInterst > 0 && isStakeMode) {
            interestToken.safeTransfer(_msgSender(), amountInterst);
        }
    }
```

RECOMMENDATION

Fix the typo.

UPDATES

- *Nov 30, 2022*: This issue has been acknowledged and fixed by the ShopNext team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Nov 26, 2022*	Private Report	Verichains Lab
1.1	*Nov 30, 2022*	Public Report	Verichains Lab

Table 2. Report versions history