# Security Audit Report

## Loopring Hebao Smart Wallet V2

SECBIT

April 14, 2023

# 1. Introduction

The Loopring Hebao V2 is a suite of smart contracts that provides the core functionality of a smart wallet. In order to assess the security of these contracts, SECBIT Labs conducted an audit from January 3 to February 15, 2023, which focused on identifying code bugs, logic flaws, and potential security risks.

Overall, the audit found that the Loopring Hebao V2 contracts do not pose any critical security risks. However, the SECBIT team did identify some areas where improvements could be made to the logical implementation, risk assessment, and code revision process (see Part 4 for details).

| Type | Description | Level | Status |
| --- | --- | --- | --- |
| Gas Optimization | 4.3.1 Gas Optimization in `approveToken()` Function | Info | To be fixed |
| Gas Optimization | 4.3.2 Gas Optimization in `_approveInternal()` Function | Info | To be fixed |
| Design & Implementation | 4.3.3 Discussion of Wallet Guardian Adding Strategy | Info | No action required |
| Design & Implementation | 4.3.4 Discussion of the `validSince` Parameter | Info | To be fixed |
| Design & Implementation | 4.3.5 Restricting the `gasToken` Type in `MetaTxLib.sol` | Info | No action required |
| Design & Implementation | 4.3.6 Discussion of the Design of `_availableQuota()` | Info | No action required |

| | | | |
|---|---|---|---|
| Design & Implementation | 4.3.7 Discussion of the Design of `isImplementationContract` | Info | No action required |
| Design & Implementation | 4.3.8 Discussion of the Parameter `quota` | Info | No action required |
| Design & Implementation | 4.3.9 Discussion of Funding Sources for Wallet Initialization | Info | No action required |
| Design & Implementation | 4.3.10 Potential Risk of Price Manipulation | Medium | To be fixed |
| Design & Implementation | 4.3.11 Potential Risk of Token Price Calculation | Low | To be fixed |
| Design & Implementation | 4.3.12 Incorrect Usage of Kyber Network Interface | Medium | To be fixed |
| Design & Implementation | 4.3.13 Potential Bypass of General Token Call Limit in `_callContractInternal()` Function | Low | To be fixed |
| Design & Implementation | 4.3.14 Potential Invalid General Call Restrictions for Tokens with `_callContractInternal()` Function | Medium | To be fixed |
| Design & Implementation | 4.3.15 Missing Check for External Calls in `ERC20SafeTransfer.sol` | Info | To be fixed |

# 2. Contract Information

This section provides a brief overview of the basic contract information and code structure of the Loopring Hebao V2 smart contracts. The information presented here is intended to give readers a high-level understanding of the code and help them navigate the codebase more efficiently.

## 2.1 Basic Information

The basic information about the Loopring Hebao V2 is shown below:

- Smart contract code
  - final review commit *a828c2ab*

## 2.2 Contract List

The following content shows the contracts that were audited by the SECBIT team in the Loopring Hebao V2:

| Name | Lines | Description |
| --- | --- | --- |
| ApprovalLib.sol | - | A utility library for better handling of signed wallet requests. |
| ERC20Lib.sol | - | A library contract that handles token transfers and contract calls. |
| ERC1271Lib.sol | - | Library contract that verify signatures. |
| GuardianLib.sol | - | Library contract for handling guardians. |

| | | |
|---|---|---|
| InheritanceLib.sol | - | Library contract for setting inheritors. |
| LockLib.sol | - | Library contract for setting the state of the wallet. |
| MetaTxLib.sol | - | A module to support wallet meta-transactions. |
| QuotaLib.sol | - | A store that maintains a daily spending quota for each wallet. |
| RecoverLib.sol | - | A contract to recover a wallet by setting a new owner and guardians. |
| UpgradeLib.sol | - | A contract to update the master copy address to a new one. |
| Utils.sol | - | A library contract to verify the validity of an address. |
| WalletData.sol | - | A contract to record the data structure of the wallet. |
| WhitelistLib.sol | - | A store that maintains a wallet's whitelisted addresses. |
| DelayedImplementationManager.sol | - | A contract that allows the proxy owner to upgrade the current version of the proxy. |
| ForwardProxy.sol | - | A forward proxy contract that works with DelayedImplementationManager.sol. |
| OfficialGuardian.sol | - | An official guardian contract. |
| SmartWallet.sol | - | The main smart wallet contract. |

| | | |
|---|---|---|
| WalletDeploymentLib.sol | - | Functionality to compute wallet addresses and deploy wallets. |
| WalletFactory.sol | - | A factory contract to create a new wallet by deploying a proxy contract. |
| AggregationalPriceOracle.sol | - | A contract for multiple oracle price aggregation. |
| CachedPriceOracle.sol | - | A contract that temporarily stores obtained prices for use. |
| KyberNetworkProxy.sol | - | A contract for obtaining prices from Kyber Network. |
| UniswapV2PriceOracle.sol | - | A contract that returns the value in Ether for any given ERC20 token. |

*Note: All files under the directory* `hebao_v2/lib/*` *have been audited in previous versions and are not listed separately here.*

# 3. Contract Analysis

This part provides details on the code assessment, including role classification and functional analysis.

## 3.1 Role Classification

Two key roles in the Loopring Hebao are Owner Account and Guardian Account.

- Owner Account
    - Description

    The smart wallet owner.

  - Authority
    - Change the master copy address
    - Add or remove guardians
    - Reset guardians
    - Set inheritor
    - Lock the smart wallet
    - Change daily quota
    - Execute meta transactions
    - Manage addresses in the whitelist
    - Transfer tokens or Ether
    - Call contracts
  - Method of Authorization

    The smart wallet owner is the initial owner of the wallet or is authorized by transferring the owner account. To enhance security, some operations require signed approval from the majority of administrative addresses, including guardians.

- Guardian Account
  - Description

    The guardian account assists the wallet owner in managing their wallet.

  - Authority
    - Add or remove guardians
    - Reset guardians
    - Change daily quota
    - Manage addresses in the whitelist
    - Transfer tokens or Ether
    - Call contracts
  - Method of Authorization

    The guardian account is authorized by the smart wallet owner. To enhance security, some operations require signed approval from the majority of administrative addresses, including the wallet owner.

## 3.2 Functional Analysis

The Loopring Hebao V2 is designed to enhance the security of wallet operations through a collaborative management model that allows multiple addresses to manage the wallet. The SECBIT team conducted a comprehensive audit of the protocol's smart contracts. We can divide the critical functions of the contracts into two parts:

### SmartWallet

The SmartWallet contract is the core logic and main entrance of the Loopring Hebao V2 protocol, providing full functionality for managing funds. The main functions of the contract are as follows:

- `addGuardian()` & `addGuardianWA()`

  These functions allow guardians to be added to the wallet.

- `removeGuardian()` & `removeGuardianWA()`

  Only the owner and guardians can remove old guardians through these functions.

- `resetGuardians()` & `resetGuardiansWA()`

  These functions allow for the resetting of all guardians.

- `changeDailyQuota()` & `changeDailyQuotaWA()`

  The owner and guardians can change the wallet's daily quota.

- `executeMetaTx()`

  A user with an Owner signature can call this function to complete the expected operation. The gas consumed by this user will be compensated by the Ether under the contract.

- `addToWhitelist()` & `addToWhitelistWA()`

  These two functions can add a whitelisted address to the wallet. Using the `addToWhitelist()` function requires waiting before it takes effect, while calling `addToWhitelistWA()` takes effect immediately.

- `transferToken()` & `transferTokenWA()`

These functions allow for the transfer of tokens or ethers from the contract.

- `callContract()` & `callContractWA()`

  The owner and guardians can use these functions to call the specified address.

### AggregationalPriceOracle, CachedPriceOracle, KyberNetworkPriceOracle, and UniswapV2PriceOracle

These oracles are provided by the protocol to evaluate the value of a given number of tokens in Ether, without involving an actual token exchange. The main function of these contracts is:

- `tokenValue()`

  This function is used to calculate the amount of Ether that can be exchanged for a specified amount of token.

# 4. Audit Detail

This section provides an overview of the audit process, and the detailed audit results demonstrate any issues and potential risks.

## 4.1 Audit Process

The audit followed SECBIT Lab's audit specification and involved an analysis of the project's code for bugs, logical implementation, and potential risks. The process consisted of four steps:

- A line-by-line analysis of the contract code
- Evaluation of vulnerabilities and potential risks in the contract code
- Communication on assessment and confirmation
- Audit report writing

## 4.2 Audit Result

After using open-source tools, including Mythril, Slither, SmartCheck, and Securify, and scanning with SECBIT Labs' internal tools, including adelaide, sf-checker, and badmsg.sender, the auditing team performed a manual assessment of the code. The results were categorized as follows:

| Number | Classification | Result |
|---|---|---|
| 1 | Normal functioning of features defined by the contract | ✓ |
| 2 | No obvious bug (e.g., overflow, underflow) | ✓ |
| 3 | Pass Solidity compiler check with no potential error | ✓ |
| 4 | Pass common tools check with no obvious vulnerability | ✓ |
| 5 | No obvious gas-consuming operation | ✓ |
| 6 | Meet with ERC20 standard | ✓ |
| 7 | No risk in low-level call (call, delegatecall, callcode) and in-line assembly | ✓ |
| 8 | No deprecated or outdated usage | ✓ |
| 9 | Explicit implementation, visibility, variable type, and Solidity version number | ✓ |
| 10 | No redundant code | ✓ |
| 11 | No potential risk manipulated by timestamp and network environment | ✓ |
| 12 | Explicit business logic | ✓ |
| 13 | Implementation consistent with annotation and other info | ✓ |

| | | |
|---|---|---|
| 14 | No hidden code about any logic that is not mentioned in the design | ✓ |
| 15 | No ambiguous logic | ✓ |
| 16 | No risk threatening the developing team | ✓ |
| 17 | No risk threatening exchanges, wallets, and DApps | ✓ |
| 18 | No risk threatening token holders | ✓ |
| 19 | No privilege on managing others' balances | ✓ |
| 20 | No non-essential minting method | ✓ |
| 21 | Correct managing hierarchy | ✓ |

## 4.3 Issues

### 4.3.1 Gas Optimization in `approveToken()` Function

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Gas Optimization | Info | Gas cost | To be fixed |

**Description**

In `ERC20Lib.sol`, the `approveToken()` function allows a wallet to authorize an allowance for a specified `to` address. The `additionalAllowance` parameter specifies the amount of funds to be added to the `to` address. If the `to` address is not whitelisted or if the condition `forceUseQuota == true`, the allowance accumulates the total amount currently used by the wallet.

However, the checkAndAddToSpent() function is executed even when additionalAllowance is zero, which wastes gas.

To optimize gas costs, we recommend adding a check on the additionalAllowance parameter and skipping the execution of the checkAndAddToSpent() function when it is zero. This will prevent unnecessary gas consumption and improve the overall efficiency of the function.

```
function approveToken(
        Wallet      storage wallet,
        PriceOracle         priceOracle,
        address             token,
        address             to,
        uint                amount,
        bool                forceUseQuota
        )
        external
    {
        uint additionalAllowance = _approveInternal(token, to,
amount);

        //@audit If the amount added is zero,
        //        the following code does not need to be
executed
        if (forceUseQuota || !wallet.isAddressWhitelisted(to))
{
            wallet.checkAndAddToSpent(priceOracle, token,
additionalAllowance);
        }
    }
```

The same situation exists in the approveThenCallContract() function.

```
function approveThenCallContract(
        Wallet  storage  wallet,
        PriceOracle      priceOracle,
        address          token,
        address          to,
```

```
        uint                amount,
        uint                value,
        bytes     calldata data,
        bool                forceUseQuota
        )
        external
        returns (bytes memory returnData)
    {
        uint additionalAllowance = _approveInternal(token, to,
amount);

        if (forceUseQuota || !wallet.isAddressWhitelisted(to))
{
            //@audit it is recommended to add the judgment
            wallet.checkAndAddToSpent(priceOracle, token,
additionalAllowance);
            wallet.checkAndAddToSpent(priceOracle, address(0),
value);
        }

        return _callContractInternal(to, value, data,
priceOracle);
    }
```

### Suggestion

The recommended modifications are as follows.

```
function approveToken(
        Wallet      storage wallet,
        PriceOracle         priceOracle,
        address             token,
        address             to,
        uint                amount,
        bool                forceUseQuota
        )
        external
    {
```

```
        uint additionalAllowance = _approveInternal(token, to,
amount);

        //@audit add this code
        if(additionalAllowance > 0){
            if (forceUseQuota ||
!wallet.isAddressWhitelisted(to)) {
            wallet.checkAndAddToSpent(priceOracle, token,
additionalAllowance);
           }
        }
    }
```

For the approveThenCallContract() function, the suggested modification would be as follows.

```
function approveThenCallContract(
        Wallet  storage  wallet,
        PriceOracle      priceOracle,
        address          token,
        address          to,
        uint             amount,
        uint             value,
        bytes   calldata data,
        bool             forceUseQuota
        )
        external
        returns (bytes memory returnData)
    {
        uint additionalAllowance = _approveInternal(token, to,
amount);


        if (forceUseQuota || !wallet.isAddressWhitelisted(to))
{
            //@audit add the corresponding judgement
            if(additionalAllowance > 0){
                wallet.checkAndAddToSpent(priceOracle, token,
additionalAllowance);
```

```
            }
            if(value > 0){
                wallet.checkAndAddToSpent(priceOracle,
address(0), value);
            }
        }

        return _callContractInternal(to, value, data,
priceOracle);
    }
```

#### Status

To be fixed in the next version. The gas optimization issue will be resolved by avoiding additional function calls when `additionalAllowance` equals 0.

### 4.3.2 Gas Optimization in `_approveInternal()` Function

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Gas Optimization | Info | Gas cost | To be fixed |

#### Description

In `ERC20Lib.sol`, the `_approveInternal()` function is used to change the allowance of a given `spender`. However, the function unnecessarily calls the `sub` function of the `SafeMath.sol` library, even though the values of the parameters `amount` and `allowance` are already determined in advance when calculating the `spender` new allowance.

To optimize gas costs and improve the overall efficiency of the function, we recommend simplifying the logic of the `_approveInternal()` function by removing the `sub` function call. This will reduce the gas consumption of the function and make it more efficient.

```
function _approveInternal(
        address token,
```

```
        address spender,
        uint    amount
        )
        private
        returns (uint additionalAllowance)
    {
        // Current allowance
        uint allowance = ERC20(token).allowance(address(this),
spender);

        if (amount != allowance) {
            // First reset the approved amount if needed
            if (allowance > 0) {
                ERC20(token).safeApprove(spender, 0);
            }
            // Now approve the requested amount
            ERC20(token).safeApprove(spender, amount);
        }

        // If we increased the allowance, calculate by how
much
        if (amount > allowance) {
            //@audit subtract directly
            additionalAllowance = amount.sub(allowance);
        }
        emit Approved(token, spender, amount);
    }
```

### Suggestion

The modifications are as follows.

```
function _approveInternal(
        address token,
        address spender,
        uint    amount
        )
        private
        returns (uint additionalAllowance)
```

```
    {
        // Current allowance
        uint allowance = ERC20(token).allowance(address(this),
spender);

        if (amount != allowance) {
            // First reset the approved amount if needed
            if (allowance > 0) {
                ERC20(token).safeApprove(spender, 0);
            }
            // Now approve the requested amount
            ERC20(token).safeApprove(spender, amount);
        }

        // If we increased the allowance, calculate by how
much
        if (amount > allowance) {
            //@audit use '-' instead of 'sub'
            additionalAllowance = amount - allowance;
        }
        emit Approved(token, spender, amount);
    }
```

**Status**

To be fixed in the next version. The issue with repetitive logical condition checks will be resolved by upgrading Solidity to v0.8 or above and deprecating the SafeMath library.

### 4.3.3 Discussion of Wallet Guardian Adding Strategy

| Risk Type | Risk Level | Impact | Status |
| --- | --- | --- | --- |
| Design & Implementation | Info | Design logic | No action required |

### Description

In GuardianLib.sol, the addGuardiansImmediately() function is called when the wallet is initialized or recovered to add guardians. This function requires that the guardian's addresses be listed in descending order. However, subsequent additions of guardians do not follow this rule.

When adding a single guardian address using the addGuardian() function or the addGuardianWA() function, the guardian address is added directly to the end of the wallet.guardians array, which may break the original rule for storing guardian's addresses. The resetGuardians() function and the resetGuardiansWA() function also reset the guardian's addresses without requiring them to be in order.

It is unclear whether the guardian addresses need to be ordered or if this is simply a shortcut to prevent duplication of addresses. We recommend documenting the intended strategy for adding guardians and specifying whether the ordering requirement is necessary or optional.

This documentation will help ensure that future updates or changes to the GuardianLib.sol contract do not inadvertently introduce vulnerabilities or bugs due to misunderstandings about the intended behavior of the contract.

```
function addGuardiansImmediately(
        Wallet    storage wallet,
        address[] memory  _guardians
        )
        external
    {
        address guardian = address(0);
        for (uint i = 0; i < _guardians.length; i++) {
            //@audit order addresses
            require(_guardians[i] > guardian,
"INVALID_ORDERING");
            guardian = _guardians[i];
            _addGuardian(wallet, guardian, 0, true);
        }
    }
```

```
function addGuardian(
        Wallet storage wallet,
        address guardian
        )
        external
    {
        //@audit add guardian address directly
        _addGuardian(wallet, guardian,
GUARDIAN_PENDING_PERIOD, false);
    }

function addGuardianWA(
        Wallet   storage  wallet,
        bytes32           domainSeparator,
        Approval calldata approval,
        address  guardian
        )
        external
        returns (bytes32 approvedHash)
    {
        ......
        //@audit add guardian address directly
        _addGuardian(wallet, guardian, 0, true);
    }


function resetGuardians(
        Wallet    storage  wallet,
        address[] calldata newGuardians
        )
        external
    {
        Guardian[] memory allGuardians = guardians(wallet,
true);
        ......
        //@audit add guardian address directly
        for (uint j = 0; j < newGuardians.length; j++) {
```

```
                _addGuardian(wallet, newGuardians[j],
GUARDIAN_PENDING_PERIOD, false);
            }
        }

function resetGuardiansWA(
        Wallet    storage  wallet,
        bytes32            domainSeparator,
        Approval  calldata approval,
        address[] calldata newGuardians
        )
        external
        returns (bytes32 approvedHash)
    {
        ......

        removeAllGuardians(wallet);
        //@audit add guardian address directly
        for (uint i = 0; i < newGuardians.length; i++) {
            _addGuardian(wallet, newGuardians[i], 0, true);
        }
    }
```

#### Status

No fix needed. The guardian does not need to maintain order, only uniqueness to prevent duplicate addresses. The uniqueness is ensured by checking order during initialization or signature verification. The guardian set is internally maintained using an enumerable set.

### 4.3.4 Discussion of the `validSince` Parameter

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Info | Design logic | To be fixed |

**Description**

In `GuardianLib.sol`, if the wallet is initialized with more than two guardian addresses, the first two guardians will have a valid time of `validSince = block.timestamp + 1`, while the remaining guardians have a valid time of `block.timestamp + pendingPeriod` (where the `pendingPeriod` parameter is zero). This inconsistency may cause confusion and introduce unnecessary complexity to the code.

To maintain consistency in the valid time of guardians, we recommend adjusting the code to ensure that all guardians are assigned the same valid time of `validSince = block.timestamp + 1`.

By making this change, the code will be more straightforward and easier to understand, reducing the risk of introducing bugs or vulnerabilities due to misunderstandings about the intended behavior of the contract.

```
function addGuardiansImmediately(
        Wallet    storage wallet,
        address[] memory  _guardians
        )
        external
    {
        address guardian = address(0);
        for (uint i = 0; i < _guardians.length; i++) {
            require(_guardians[i] > guardian,
"INVALID_ORDERING");
            guardian = _guardians[i];
            //@audit add guardian address
            _addGuardian(wallet, guardian, 0, true);
        }
    }

function _addGuardian(
        Wallet storage wallet,
        address guardian,
        uint    pendingPeriod,
        bool    alwaysOverride
```

```
        )
        internal
    {
        uint _numGuardians = numGuardians(wallet, true);
        require(_numGuardians < MAX_GUARDIANS,
"TOO_MANY_GUARDIANS");
        require(guardian != wallet.owner,
"GUARDIAN_CAN_NOT_BE_OWNER");

        uint validSince = block.timestamp + 1;

        //@audit effective time is changed
        if (_numGuardians >= 2) {
            validSince = block.timestamp + pendingPeriod;
        }
        validSince = storeGuardian(wallet, guardian,
validSince, alwaysOverride);
        emit GuardianAdded(guardian, validSince);
    }
```

### Suggestion

The recommended modification is as follows.

```
function _addGuardian(
        Wallet storage wallet,
        address guardian,
        uint    pendingPeriod,
        bool    alwaysOverride
        )
        internal
    {
        uint _numGuardians = numGuardians(wallet, true);
        require(_numGuardians < MAX_GUARDIANS,
"TOO_MANY_GUARDIANS");
        require(guardian != wallet.owner,
"GUARDIAN_CAN_NOT_BE_OWNER");

        uint validSince = block.timestamp + 1;
```

```
        if (_numGuardians >= 2) {
            //@audit modify the following code
            validSince = block.timestamp + pendingPeriod + 1;
        }
        validSince = storeGuardian(wallet, guardian,
validSince, alwaysOverride);
        emit GuardianAdded(guardian, validSince);
    }
```

#### Status

To be fixed in the next version.

### 4.3.5 Restricting the `gasToken` Type in `MetaTxLib.sol`

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Info | Logic correctness | No action required |

#### Description

In `MetaTxLib.sol`, the `executeMetaTx()` function is called by a user authorized by `owner`. As compensation, the gas cost for calling the `executeMetaTx()` function is paid directly from the wallet.

The parameter `gasCost` represents the amount of Ether corresponding to the gas consumed by executing this function. In the `checkAndAddToSpent()` function, the code calculates the equivalent value of `gasCost` in Ether and deducts it from the wallet's `gasToken` balance, which is then transferred to the `feeRecipient` address specified by the owner.

However, the current code does not restrict the type of `gasToken`, which could result in an accidental transfer of user funds. To refine the code and reduce this risk, we recommend restricting the `gasToken` type to Ether.

By adding this restriction, the code will be more robust and less prone to errors, ensuring that the wallet's funds are used only as intended.

```
function executeMetaTx(
        Wallet      storage wallet,
        bytes32             DOMAIN_SEPARATOR,
        PriceOracle         priceOracle,
        MetaTx      memory  metaTx
        )
        public
        returns (bool success)
    {
        ......

        // Reimburse
        if (metaTx.gasPrice > 0 && (!metaTx.requiresSuccess ||
success)) {
            uint gasToReimburse = gasUsed <= metaTx.gasLimit ?
gasUsed : metaTx.gasLimit;

            //@audit calculate the amount of Ether used
            uint gasCost =
gasToReimburse.mul(metaTx.gasPrice);

            wallet.checkAndAddToSpent(
                priceOracle,
                metaTx.gasToken,  //@audit unrestricted type
of gasToken
                gasCost           //@audit amount of Ether
            );

            //@audit transfer token
            ERC20Lib.transfer(metaTx.gasToken,
metaTx.feeRecipient, gasCost);
        }

        emit MetaTxExecuted(
            metaTx.nonce,
```

```
                metaTx.nonce == 0 ? returnData.toBytes32(0) :
bytes32(0),
                metaTxHash,
                success,
                gasUsed
            );
        }
```

**Status**

No need to fix this issue. The `gasPrice` parameter passed by the official relayer specifies the token (Ether or ERC20 token), and the `gasCost` is converted using a price conversion, which means that it may not be Ether. It is only controlled and validated by the centralized backend, with no restrictions on the contract side.

### 4.3.6 Discussion of the Design of `_availableQuota()`

| Risk Type | Risk Level | Impact | Status |
| --- | --- | --- | --- |
| Design & Implementation | Info | Design logic | No action required |

**Description**

In `QuotaLib.sol`, the `_availableQuota()` function returns the amount of quota available to the wallet, and the `_currentQuota()` function returns the total amount of quota in the wallet.

In the case where the total amount of the wallet is 0, that is `quota == 0`, the `_availableQuota()` function returns the maximum amount `MAX_QUOTA`. However, this logic is unclear and may cause confusion about the intended behavior of the contract.

We recommend reviewing the design of the `_availableQuota()` function to ensure that it accurately reflects the intended behavior of the contract.

```
function _currentQuota(Quota memory q)
```

```
        private
        view
        returns (uint)
    {
        return q.pendingUntil <= block.timestamp ?
q.pendingQuota : q.currentQuota;
    }

function _availableQuota(Quota memory q)
        private
        view
        returns (uint)
    {
        uint quota = _currentQuota(q);

        //@audit notice
        if (quota == 0) {
            return MAX_QUOTA;
        }
        uint spent = _spentQuota(q);
        return quota > spent ? quota - spent : 0;
    }
```

**Status**

As stated in the code comment, a value of 0 for `newQuota` or `currentQuota` indicates that unlimited quota or daily quota is disabled. In this case, the `_availableQuota()` function returns the maximum amount `MAX_QUOTA`. This behavior correctly reflects the intended behavior of the contract, and no action is required.

### 4.3.7 Discussion of the Design of `isImplementationContract`

| Risk Type | Risk Level | Impact | Status |
| --- | --- | --- | --- |
| Design & Implementation | Info | Design logic | No action required |

**Description**

In `SmartWallet.sol`, the function `initialize()` calls the modifier function `disableInImplementationContract` when initializing the wallet, which requires the parameter `isImplementationContract` to be false. However, the parameter `isImplementationContract` is already assigned the value true in the constructor, which causes the call of the `initialize()` function to fail.

```
modifier disableInImplementationContract
    {
        require(!isImplementationContract,
"DISALLOWED_ON_IMPLEMENTATION_CONTRACT");
        _;
    }

constructor(
        PriceOracle _priceOracle,
        address     _blankOwner
        )
    {
        isImplementationContract = true;

        DOMAIN_SEPARATOR = EIP712.hash(
            EIP712.Domain("LoopringWallet", "2.0.0",
address(this))
        );

        priceOracle = _priceOracle;
        blankOwner = _blankOwner;
    }

    /// @dev Set up this wallet.
    ///
    ///      Note that calling this method more than once will
throw.
    ///
    /// @param owner The owner of this wallet, must not be
address(0).
```

```
    /// @param guardians The guardians of this wallet.
    function initialize(
        address             owner,
        address[] calldata  guardians,
        uint                quota,
        address             inheritor,
        address             feeRecipient,
        address             feeToken,
        uint                feeAmount
        )
        external
        override
        disableInImplementationContract
    {
        require(wallet.owner == address(0),
"INITIALIZED_ALREADY");
        require(owner != address(0), "INVALID_OWNER");

        ......
    }
```

### Status

After reviewing the contract, we have determined that `SmartWallet.sol` takes a proxy pattern to deploy. This means that a new proxy contract is deployed for each new user. The `disableInImplementationContract` function is designed to prevent the implementation contract of a proxy from being repeatedly initialized, which could introduce vulnerabilities or other issues.

Since the value of `isImplementationContract` is set to true in the constructor, this indicates that the implementation contract has already been initialized and that the `initialize()` function should not be called again. Therefore, the current behavior of the contract is intentional, and no action is required to address this issue.

### 4.3.8 Discussion of the Parameter `quota`

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Info | Design logic | No action required |

**Description**

In `SmartWallet.sol`, when initializing the configured wallet quota parameter in the `initialize()` function, if the `quota` parameter is zero, the quota setting will be skipped. By analyzing the `setQuota()` function, we noticed that when the quota value is `MAX_QUOTA` (`uint128(-1)`), the code will adjust the quota value to 0, which is the same as setting the quota value to 0 directly. It is important to verify whether the code should also skip the case when `quota == MAX_QUOTA`.

In addition, the maximum amount is `MAX_QUOTA` (`uint128(-1)`), and the actual argument type passed in is `uint` (default 256 bits). In this case, if the user defaults to using the maximum amount, `uint(-1)` could be passed in from the front end, and that value is greater than `uint128(-1)`, which will cause the function call to fail.

```
function initialize(
        address             owner,
        address[] calldata  guardians,
        uint                quota,
        address             inheritor,
        address             feeRecipient,
        address             feeToken,
        uint                feeAmount
        )
        external
        override
        disableInImplementationContract
    {
```

```
        require(wallet.owner == address(0),
"INITIALIZED_ALREADY");
        require(owner != address(0), "INVALID_OWNER");

        wallet.owner = owner;
        wallet.creationTimestamp = uint64(block.timestamp);
        wallet.addGuardiansImmediately(guardians);

        if (quota != 0) {
            wallet.setQuota(quota, 0);
        }

        ......
    }

//@audit located in QuotaLib.sol
uint128 public constant MAX_QUOTA = uint128(-1);
function setQuota(
        Wallet storage wallet,
        uint           newQuota,
        uint           effectiveTime
        )
        internal
    {
        //@audit new amount cannot exceed
MAX_QUOTA(uint128(-1))
        require(newQuota <= MAX_QUOTA, "INVALID_VALUE");
        if (newQuota == MAX_QUOTA) {
            newQuota = 0;
        }

        uint __currentQuota = currentQuota(wallet);
       ......
    }
```

### Status

After reviewing the code and the contract documentation, we have determined that the parameters of the `initialize()` function are set with the help of the wallet client, and not by the end user. Therefore, the client only needs to pass the parameters according to the defined rules in the contract code. As a result, no action is required to address these issues.

### 4.3.9 Discussion of Funding Sources for Wallet Initialization

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Info | Design logic | No action required |

### Description

In `SmartWallet.sol`, the `createWallet()` function in `WalletFactory.sol` is used to create a new wallet contract and initialize its base parameters. Upon analyzing the `_deploy()` function responsible for creating the wallet, it was found that the user does not transfer ERC20 tokens or Ether into the contract during the creation process. However, the wallet may require a transfer of `feeAmount` amount of `feeToken` to the specified `feeRecipient` address, which is not possible as there are no funds in the wallet at this point.

```
//@audit located in WalletFactory.sol
function createWallet(
        WalletConfig calldata config,
        uint                  feeAmount
        )
        external
        returns (address wallet)
    {
        require(feeAmount <= config.maxFeeAmount,
"INVALID_FEE_AMOUNT");
```

```
        _validateConfig(config);

        //@audit create new wallet
        wallet = _deploy(config.owner, config.salt);

        //@audit initialize wallet
        _initializeWallet(wallet, config, feeAmount);
    }

//@audit located in WalletDeploymentLib.sol
function _deploy(
        address owner,
        uint    salt
        )
        internal
        returns (address payable wallet)
    {
        wallet = Create2.deploy(
            computeWalletSalt(owner, salt),
            getWalletCode()
        );
    }

//@audit located in Create2.sol
function deploy(bytes32 salt, bytes memory bytecode) internal
returns (address payable) {
        address payable addr;
        // solhint-disable-next-line no-inline-assembly
        assembly {
            addr := create2(0, add(bytecode, 0x20),
mload(bytecode), salt)
        }
        require(addr != address(0), "CREATE2_FAILED");
        return addr;
    }

//@audit located in SmartWalle.sol
function initialize(
        address                owner,
```

```
        address[] calldata  guardians,
        uint                quota,
        address             inheritor,
        address             feeRecipient,
        address             feeToken,
        uint                feeAmount
        )
        external
        override
        disableInImplementationContract
    {
        ......

        // Pay for the wallet creation using wallet funds
        // @audit source of funds?
        if (feeRecipient != address(0) && feeAmount > 0) {
            ERC20Lib.transfer(feeToken, feeRecipient,
feeAmount);
        }
    }
```

**Status**

In the typical process, users do not directly send a transaction to create a wallet contract. Instead, they sign the wallet parameters and then create it via a transaction sent by someone else. Wallets are created by `Create2`, which makes it easy to calculate addresses in advance. Therefore, it is necessary for the user to transfer the required fees to the calculated contract address in advance, while the wallet is not yet created.

### 4.3.10 Potential Risk of Price Manipulation

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Medium | Price oracle | To be fixed |

### Description

In `AggregationalPriceOracle.sol`, the `tokenValue()` function returns the number of `token` valued in Ethers for the specified `amount` quantity by reading return values from multiple oracles and calculating the average of these returns. While this design seeks to reduce the risk of possible deviations from normal prices or artificial manipulation, it does not completely eliminate this risk.

If a user manipulates one or more of these oracles before using the function, the result obtained by the `tokenValue()` function will still deviate from the actual price. Since it is not clear what type of oracles will be used in production, it is impossible to determine whether the above assumptions are realized or not.

The price data is mainly used to calculate the quote limit. Therefore, the team should carefully consider whether oracle price manipulation has a critical impact on the function and its usage. If it does, we recommend improving the oracle module, for example, by using Time-Weighted Average Price (TWAP) Oracles or other reliable services, to reduce the risk of possible price manipulation and ensure the reliability of the price data used by the contract.

```
function tokenValue(address token, uint amount)
        public
        view
        override
        returns (uint)
    {
        uint total;
        uint count;
        for (uint i = 0; i < oracles.length; i++) {
            uint value =
PriceOracle(oracles[i]).tokenValue(token, amount);
            if (value > 0) {
                count += 1;
                total = total.add(value);
            }
        }
```

```
        //@audit calculate the average value of Ether
        return count == 0 ? 0 : total / count;
    }
```

**Status**

To be fixed in the next version. Aggregating all oracle price data using the average method may still be susceptible to interference from anomalous data. Attackers can alter the final aggregated price data by manipulating a single oracle. Therefore, the next version will use the median method to aggregate price data, while excluding data with abnormal changes relative to historical prices. For Uniswap price data, the contract will abandon the direct calculation of spot price using the y/x method and use the TWAP data recommended by the audit.

### 4.3.11 Potential Risk of Token Price Calculation

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Low | Design logic | To be fixed |

**Description**

The `CachedPriceOracle.sol` contract reads the price data of a given oracle, stores it, and uses it for a specified period (currently seven days). However, this code logic has several potential issues.

First, the price data may become outdated, and users could use this outdated data when calling the `tokenValue()` function, if no one triggers a price update on time.

Second, there is a risk of price manipulation, since the `updateTokenPrice()` function, which obtains and stores the price data from a specified oracle, is a public function with no permission restrictions. This means any user can call it to update the price data.

Third, when the price expires, the `tokenValue()` function returns a zero value, which may be risky for external contracts that use this price directly.

Therefore, it should be clarified whether `CachedPriceOracle.sol` will be used for critical logic. If so, we recommend improving the contract to reduce the risk. This could include implementing more sophisticated oracle mechanisms, such as Time-Weighted Average Price (TWAP) Oracles, using access control to restrict who can update the price data, and improving the handling of expired prices.

```
function updateTokenPrice(
        address token,
        uint    amount
        )
        external
        returns (uint value)
    {
        value = oracle.tokenValue(token, amount);
        if (value > 0) {
            _cacheTokenPrice(token, amount, value);
        }
    }

function _cacheTokenPrice(
        address token,
        uint    amount,
        uint    value
        )
        internal
    {
        prices[token].amount = amount.toUint128();
        prices[token].value = value.toUint96();
        prices[token].timestamp = block.timestamp.toUint32();
        emit PriceCached(token, amount, value,
block.timestamp);
    }

    function tokenValue(address token, uint amount)
        public
```

```
            view
            override
            returns (uint)
        {
            TokenPrice memory tp = prices[token];
            if (tp.timestamp > 0 && block.timestamp < tp.timestamp
+ EXPIRY_PERIOD) {
                return uint(tp.value).mul(amount) / tp.amount;
            } else {
                return 0;
            }
        }
```

#### Status

To be fixed in the next version. The function `updateTokenPrice()` can be called by anyone and a large `amount` value can cause the updated price to deviate from the normal value. In the next version, this function will be used internally and automatically updated when the cache expires and a new price is required.

### 4.3.12 Incorrect Usage of Kyber Network Interface

| Risk Type | Risk Level | Impact | Status |
| --- | --- | --- | --- |
| Design & Implementation | Medium | Design logic | To be fixed |

#### Description

The `KyberNetworkPriceOracle.sol` contract calls the `getExpectedRate()` function of the Kyber Network to obtain the token price. However, by analyzing the `getExpectedRate()` function, we find that the first return value indicates the number of Ether that can be exchanged per unit of a token, not the number of Ether that can be exchanged for the `amount` of `token`. Therefore, the current implementation may be incorrect.

The relevant line of code can be found here: https://etherscan.io/address/0x7C66550C9c730B6fdd4C03bc2e73c5462c5F7ACC#code#L1534.

The team should review and update the implementation of the `KyberNetworkPriceOracle.sol` contract to ensure that it uses the correct return value from the `getExpectedRate()` function or another function that provides the desired information. This is essential to ensure the accuracy of the token price and the overall security of the smart contract.

```
function tokenValue(address token, uint amount)
        public
        view
        override
        returns (uint value)
    {
        if (amount == 0) return 0;
        if (token == address(0) || token == ethTokenInKyber) {
            return amount;
        }
        (value,) = kyber.getExpectedRate(
            ERC20(token),
            ERC20(ethTokenInKyber),
            amount
        );
    }
```

```
//@audit
https://etherscan.io/address/0x7C66550C9c730B6fdd4C03bc2e73c54
62c5F7ACC#code#L1534
/// @notice Backward compatible API
/// @dev Gets the expected and slippage rate for exchanging
src -> dest token
/// @dev worstRate is hardcoded to be 3% lower of expectedRate
/// @param src Source token
/// @param dest Destination token
/// @param srcQty Amount of src tokens in twei
/// @return expectedRate for a trade after deducting network
fee.
```

```
/// @return worstRate for a trade. Calculated to be
expectedRate * 97 / 100
function getExpectedRate(
    ERC20 src,
    ERC20 dest,
    uint256 srcQty
) external view returns (uint256 expectedRate, uint256
worstRate) {
    if (src == dest) return (0, 0);
    uint256 qty = srcQty & ~PERM_HINT_GET_RATE;

    TradeData memory tradeData = initTradeInput({
        trader: payable(address(0)),
        src: src,
        dest: dest,
        srcAmount: (qty == 0) ? 1 : qty,
        destAddress: payable(address(0)),
        maxDestAmount: 2**255,
        minConversionRate: 0,
        platformWallet: payable(address(0)),
        platformFeeBps: 0
    });

    tradeData.networkFeeBps = getNetworkFee();

    (, expectedRate) = calcRatesAndAmounts(tradeData, "");

    worstRate = (expectedRate * 97) / 100; // backward
compatible formula
}
```

### Status

To be fixed in the next version. The contract will calculate `ExpectedRate * amount` as the actual token value.

### 4.3.13 Potential Bypass of General Token Call Limit in `_callContractInternal()` Function

| Risk Type | Risk Level | Impact | Status |
| --- | --- | --- | --- |
| Design & Implementation | Low | Arbitrary call | To be fixed |

**Description**

The `_callContractInternal()` function disallows general calls to token contracts when `priceOracle` is not the zero address. However, when this function is called in the `callContractWA()` and `approveThenCallContractWA()` functions, a zero address is used for `priceOracle`. This means that the general call limit could be bypassed in these cases.

To ensure the security of the smart contract, we recommend that the team review the design of the smart contract to ensure that this potential bypass is covered. Possible solutions could include updating the `_callContractInternal()` function to account for the zero address for `priceOracle` in the `callContractWA()` and `approveThenCallContractWA()` functions or implementing additional checks to prevent unauthorized general calls.

By implementing these changes, the security of the smart contract will be improved and the risk of potential unauthorized general calls will be reduced.

```
    // @audit located in ERC20Lib.sol
    function callContractWA(
        Wallet   storage  wallet,
        bytes32           domainSeparator,
        Approval calldata approval,
        address           to,
        uint              value,
        bytes    calldata data
        )
```

```
        external
        returns (bytes32 approvedHash, bytes memory
returnData)
    {
        approvedHash = wallet.verifyApproval(
            domainSeparator,
            SigRequirement.MAJORITY_OWNER_REQUIRED,
            approval,
            abi.encode(
                CALL_CONTRACT_TYPEHASH,
                approval.wallet,
                approval.validUntil,
                to,
                value,
                keccak256(data)
            )
        );

        returnData = _callContractInternal(to, value, data,
PriceOracle(0)); // @audit could be used to call tokens
because oracle is zero
    }

    function _callContractInternal(
        address              to,
        uint                 value,
        bytes       calldata txData,
        PriceOracle          priceOracle
        )
        private
        returns (bytes memory returnData)
    {
        require(to != address(this), "SELF_CALL_DISALLOWED");

        if (priceOracle != PriceOracle(0)) {
            if (txData.length >= 4) {
                bytes4 methodId = txData.toBytes4(0);
                //
bytes4(keccak256("transfer(address,uint256)")) = 0xa9059cbb
```

```
                //
bytes4(keccak256("approve(address,uint256)")) = 0x095ea7b3
                if (methodId == bytes4(0xa9059cbb) ||
                    methodId == bytes4(0x095ea7b3)) {
                    // Disallow general calls to token
contracts (for tokens that have price data
                    // so the quota is actually used).
                    require(priceOracle.tokenValue(to, 1e18)
== 0, "CALL_DISALLOWED");
                }
            }
        }

        bool success;
        (success, returnData) = to.call{value: value}(txData);
        require(success, "CALL_FAILED");

        emit ContractCalled(to, value, txData);
    }
```

**Status**

To be fixed in the next version. The issue of bypassing call limit when the `priceOracle` is 0 will be prevented.

### 4.3.14 Potential Invalid General Call Restrictions for Tokens with `_callContractInternal()` Function

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Medium | Bypass restrictions | To be fixed |

### Description

The `_callContractInternal()` function disallows general calls to token contracts when calling typical `transfer()` and `approve()` functions. However, this approach may not be sufficient for all tokens. For example, some tokens adopt the `increaseAllowance()` interface, which can also be used for increasing token allowance.

To ensure that the smart contract can interact with all tokens securely, we recommend that the team review the general call restrictions implemented in the `_callContractInternal()` function and ensure that they are sufficient for all tokens that the smart contract may interact with. If additional restrictions are needed to prevent unauthorized general calls, appropriate measures should be taken to implement these restrictions.

By making these changes, the security of the smart contract will be improved and the risk of potential unauthorized general calls will be reduced, ensuring that the smart contract can interact with all tokens securely.

```
    // @audit located in ERC20Lib.sol
    function _callContractInternal(
        address                   to,
        uint                      value,
        bytes            calldata txData,
        PriceOracle               priceOracle
        )
        private
        returns (bytes memory returnData)
    {
        require(to != address(this), "SELF_CALL_DISALLOWED");

        if (priceOracle != PriceOracle(0)) {
            if (txData.length >= 4) {
                bytes4 methodId = txData.toBytes4(0);
                //
bytes4(keccak256("transfer(address,uint256)")) = 0xa9059cbb
                //
bytes4(keccak256("approve(address,uint256)")) = 0x095ea7b3
```

```
                if (methodId == bytes4(0xa9059cbb) ||
                    methodId == bytes4(0x095ea7b3)) { //
@audit what about other methods, like transferFrom or
increaseAllowance?
                    // Disallow general calls to token
contracts (for tokens that have price data
                    // so the quota is actually used).
                    require(priceOracle.tokenValue(to, 1e18)
== 0, "CALL_DISALLOWED");
                }
            }
        }

        bool success;
        (success, returnData) = to.call{value: value}(txData);
        require(success, "CALL_FAILED");

        emit ContractCalled(to, value, txData);
    }
```

### Status

To be fixed in the next version. The contract will add the necessary method id checks, such as `increaseAllowance()` or `transferFrom()`, to prevent bypassing the limit check through general calls.

### 4.3.15 Missing Check for External Calls in `ERC20SafeTransfer.sol`

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Low | Design logic | To be fixed |

**Description**

The `ERC20SafeTransfer.sol` contract provides a fix to call token contracts with return-value incompatibility by using a low-level call like `token.call{gas: gasLimit}(callData)`. However, it does not check if the `token` address is a smart contract. Therefore, if someone passes a non-contract address as `token`, this function will not detect it and will never throw an exception. This introduces potential risks and could be used to bypass other checks in certain cases.

To prevent this issue, it is recommended to update the implementation of the `ERC20SafeTransfer.sol` contract to include a check that ensures that the `token` address is a smart contract. For example, the OpenZeppelin team's implementation of [`SafeERC20.sol`] includes a similar check. This will help to ensure the security of the smart contract and protect it from potential risks associated with external calls.

```
    // @audit located in ERC20SafeTransfer.sol
    function safeTransferFromWithGasLimit(
        address token,
        address from,
        address to,
        uint    value,
        uint    gasLimit
        )
        internal
        returns (bool)
    {
        //
bytes4(keccak256("transferFrom(address,address,uint256)")) =
0x23b872dd
        bytes memory callData = abi.encodeWithSelector(
            bytes4(0x23b872dd),
            from,
            to,
            value
        );
```

```
        (bool success, ) = token.call{gas: gasLimit}
(callData); // @audit should check if token isContract
        return checkReturnValue(success);
    }

    function safeTransferWithGasLimit(
        address token,
        address to,
        uint    value,
        uint    gasLimit
        )
        internal
        returns (bool)
    {
        // bytes4(keccak256("transfer(address,uint256)")) =
0xa9059cbb
        bytes memory callData = abi.encodeWithSelector(
            bytes4(0xa9059cbb),
            to,
            value
        );
        (bool success, ) = token.call{gas: gasLimit}
(callData); // @audit should check if token isContract
        return checkReturnValue(success);
    }
```

### Status

To be fixed in the next version. The contract will add a check for whether the token address is a contract address to prevent normal execution when the address is an EOA address.

# 5. Conclusion

SECBIT Labs recently conducted an audit and analysis of the Loopring Hebao V2 smart contracts and identified several issues that could be optimized. The findings and recommendations are listed above and aim to improve the security and overall quality of the smart contracts. We recommend that the Loopring team carefully review and consider these suggestions to help ensure the security and integrity of the platform.

# Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

# APPENDIX

## Vulnerability/Risk Level Classification

| Level | Description |
| --- | --- |
| High | Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock assets inside the contract. |
| Medium | Damage contract's security under given conditions and cause impairment of benefit for stakeholders. |
| Low | Cause no actual impairment to contract. |
| Info | Relevant to practice or rationality of the smart contract, could possibly bring risks. |

**SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.**


https://secbit.io

audit@secbit.io

@secbit_io