# Security Audit Report

## Solv Protocol (V2 IVO)

SECBIT

Jan 26, 2022

# 1. Introduction

The Solv Protocol is the decentralized platform for creating, managing, and trading Financial NFTs. SECBIT Labs conducted an audit from January 10th to January 26th, 2022, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the Solv Protocol contract has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising(see part 4 for details).

| Type | Description | Level | Status |
|---|---|---|---|
| Design & Implementation | 4.3.1 There is a logical error in the code that handles the buyer's payment commission in the `buyByAmount()` function. | Medium | Fixed |
| Design & Implementation | 4.3.2 There is a logical problem when the buyer does not transfer enough ether to buy the specified number of units. | Medium | Fixed |
| Design & Implementation | 4.3.3 There is a logical problem with the way the `buyByUnits()` function calculates the commission when the buyer transfers a quantity of ether exactly equal to the value of the quantity of `units` he specifies. | Medium | Fixed |
| Design & Implementation | 4.3.4 The `_mintVoucher()` function needs to adjust the corresponding authorization amount so that the transfer amount matches the authorization amount. | Info | Fixed |
| Design & Implementation | 4.3.5 The function `mint()` without permission protection may have unexpected consequences. | Info | Discussed |
| Design & Implementation | 4.3.6 Discussion of the logic of the `split()` code. | Info | Discussed |
| Design & Implementation | 4.3.7 Adds the `_afterTransferUnits()` function to improve the capability of the `_transferUnitsFrom()` function. | Info | Discussed |

# 2. Contract Information

This part describes the basic contract information and code structure.

### 2.1 Basic Information

The basic information about the Solv Protocol contract is shown below:

- Project website
  - https://solv.finance
- Smart contract code
  - https://github.com/solv-finance/solv-v2-ivo
    - initial review commit *bf6dac3*
    - final review commit *b207d5e*

### 2.2 Contract List

The following content shows the contracts included in the Solv protocol, which the SECBIT team audits:

| Name | Lines | Description |
|---|---|---|
| AdminControl.sol | 26 | An abstract contract initializes and modifies the administrator's address. |
| ERC20TransferHelper.sol | 86 | A library contract that handles funds transfer. |
| VNFTTransferHelper.sol | 97 | A library contract that handles the transfer of VNFT tokens. |
| Solver.sol | 43 | The administrator can control the invocation of crucial functions through this contract. |
| ISolvMarketplace.sol | 95 | Interface contract for `SolvConvertibleMarket.sol`. |
| SolvConvertibleMarket.sol | 691 | Users can use this contract to sell their owned vouchers. |
| InitialConvertibleOfferingMarket.sol | 142 | The issuer offers a token contract, and the token sold |

| | | |
|---|---|---|
| | | will be locked in for a specified period before it is sold at a settlement price. |
| OfferingMarketCore.sol | 584 | The core contract that handles the offering of tokens, which will be inherited. |
| PriceManager.sol | 86 | Auxiliary contract to record the price of units. |
| InitialVestingOfferingMarket.sol | 179 | This contract provides the issuer with three ways of releasing tokens. |
| ConvertiblePool.sol | 425 | A contract to store the purchaser's funds and the issuer's refunded funds. The issuer can retrieve the remaining token through this contract. |
| ConvertibleVoucher.sol | 175 | Upon expiry, the purchaser buys the token sold by the issuer at the settlement price. |
| ChainlinkPriceOracle.sol | 180 | Use the ChainLink oracle to provide the average price of a token to be sold over a specified period. |
| ManualPriceOracle.sol | 57 | Contracts for which the administrator sets the settlement price manually. |
| UniswapV2PriceOracle.sol | 308 | Use the UniswapV2 oracle to provide the average price of a token to be sold over a specified period. |
| PriceOracleManager.sol | 144 | Administrators can switch to different oracles to obtain contracts at settlement prices. |
| FlexibleDateVestingPool.sol | 298 | The contract stores the token to be released. |
| FlexibleDateVestingVoucher.sol | 163 | The user claims the released token through this contract. |
| VNFTCoreV2.sol | 283 | This contract implements the merging and splitting of VNFT tokens on the basic ERC721 functionality. |
| VoucherCore.sol | 151 | A peripheral contract to the VNFT token. |

## 3. Contract Analysis

This part describes code assessment details, including two items: "role classification" and "functional analysis".

## 3.1 Role Classification

There are two key roles in Solv Protocol: Governance Account and Common Account.

- Governance Account
  - Description

    Contract administrator
  - Authority
    - Update basic parameters
    - Transfer ownership
  - Method of Authorization

    The contract administrator is the contract's creator or authorized by the transferring of governance account.
- Common Account
  - Description

    Sell or buy vouchers
  - Authority
    - Sell ERC20 token as a VNFT token
    - Buy ERC20 tokens, which will be transferred to the user in the form of a VNFT token
  - Method of Authorization

    No authorization required

## 3.2 Functional Analysis

The Solv Protocol provides Financial NFTs with a comprehensive solution for their creation, management, and transactions. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. We can divide the critical functions of the contract into several parts:

### SolvConvertibleMarket

This contract offers two ways to sell vouchers: at a fixed price or a variable price. When selling vouchers, this contract also provides purchasers the option to buy in funds or units.

The main functions in `SolvConvertibleMarket` are as below:

- `publishFixedPrice()`

  Sellers sell their VNFT token holdings for a fixed price.
- `publishDecliningPrice()`

  Sellers sell their VNFT token holdings at a variable price.

- `buyByAmount()`

  Buyers provide a specified amount of money to purchase the voucher.
- `buyByUnits()`

  The buyer specifies the number of units they wish to purchase. These units will be packaged as a VNFT token and transferred to the purchaser.
- `remove()`

  The seller retrieves the voucher, which he has not sold.

### InitialConvertibleOfferingMarket

The issuer will pre-sell ERC20 tokens at a specified price and sell them at a settlement price after the expiry time.

The main functions in `InitialConvertibleOfferingMarket` are as below:

- `offer()`

  The issuer offers a voucher. The issuer will need to transfer the ERC20 token to be sold to this contract.
- `_mintVoucher()`

  This function encapsulates the underlying token purchased by the buyer in the form of a VNFT token.
- `_refund()`

  The issuer retrieves the unsold ERC20 token.

### OfferingMarketCore

This contract implements the core function for the issuer to sell vouchers. The main functions in `OfferingMarketCore` are as below:

- `_offer()`

  This function records the detailed parameters of the voucher sold by the issuer, such as offeringId, startTime, endTime, etc.
- `buy()`

  The buyer purchases the voucher in units. This function will mint a new VNFT token based on the number of units purchased by the buyer.
- `remove()`

  The issuer withdraws the remaining vouchers before the start or after the end of the presale.

### InitialVestingOfferingMarket

This contract provides three ways to unlock the voucher: LINEAR, ONE_TIME and STAGED. The main functions in `InitialVestingOfferingMarket` are as below:

- `offer()`

  This function records the detailed parameters of the voucher sold by the issuer, such as offeringId, startTime, endTime, etc.
- `_mintVoucher()`

  This function encapsulates the underlying token purchased by the buyer in the form of a VNFT token.
- `_refund()`

  The issuer retrieves the unsold ERC20 token.

### PriceOracleManager

This contract provides average price data for voucher maturity settlement. The main functions in `PriceOracleManager` are as below:

- `refreshUnderlyingPriceOfMaturity()`

  Anyone can call this function to update to get the settlement price.
- `getPriceOfMaturity()`

  This function provides the settlement price to Solv Protocol, which will calculate the actual ERC20 token received by the user based on this function.

### ConvertiblePool

This contract primarily handles voucher maturity settlement tasks. In addition, this contract manages the underlying token and currency allocations according to the settlement price.

The main functions in `ConvertiblePool` are as below:

- `createSlot()`

  Create slot function. The slot acts as a special attribute of the VNFT token. It marks the uniqueness of the user operation information.
- `refund()`

  The issuer refunds the purchaser's funds by the maturity time.
- `withdraw()`

After the voucher matures, the issuer gets back the remaining ERC20 token (underlying token).

- `claim()`

This function allows convertible voucher holders to claim fund currency or underlying token after maturity.

### FlexibleDateVestingPool

This contract handles the release of the underlying token. The main functions in `FlexibleDateVestingPool` are as below:

- `mint()`

  Transfer the voucher purchased by the user to the current contract address.
- `claim()`

  This function retrieves the underlying token that the buyer has currently unlocked in the form of a VNFT token.

### VoucherCore

This contract is an implementation contract for the VNFT token. It combines the descriptive properties of the ERC-721 token with the liquidity characteristics of the ERC-20 token.

The main functions in `VoucherCore` are as below:

- `split()`

  Allows users to split the VNFT token they hold and obtain a new VNFT token. The newly generated VNFT token has the same slot as the original VNFT token.
- `merge()`

  Allows users to merge multiple VNFT tokens with the same slot into a single VNFT token. The source VNFT will be burned before the merge.
- `_mint()`

The user calls this function to mint a new VNFT token.

- `burn()`

  The user can call this function to burn the VNFT token it holds.

# 4. Audit Detail

This part describes the process, and the detailed results of the audit also demonstrate the problems and potential risks.

## 4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bug, logical implementation, and potential risks. The process consists of four steps:

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

## 4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into the following types:

| 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 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 There is a logical error in the code that handles the buyer's payment commission in the `buyByAmount()` function.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Medium | Design logic | Fixed |

**Description**

This contract provides sellers with `FIXED` and `DECLIINING_BY_TIME` options to sell their voucher holdings. The contract also provides buyers with different strategies for buying vouchers by the number of funds and units. Buyers can purchase vouchers for a given `amount_` of funds by calling the `buyByAmount()` function. In addition, the function will calculate the platform fee that the buyer needs to pay based on the number of funds they have provided. The seller can choose whether the buyer pays the fee or the seller pays it.

Assuming that the buyer pays the fee, the code uses the amount of `(amount_ - fee)` to calculate the amount of `units` that the buyer will receive. The fee is included in the parameter `amount_`. However, when the code calculates the number of funds the buyer needs to transfer, it requires the buyer to transfer `(amount_ + fee)`, which means that the parameter `amount_` is the number of funds the buyer will spend on the voucher after the fee is removed. There is a contradiction in logic.

```
// @audit located in SolvConvertibleMarket.sol
function buyByAmount(uint24 saleId_, uint256 amount_)
        external
        payable
        virtual
        override
        returns (uint128 units_)
    {
        Sale storage sale = sales[saleId_];
        address buyer = msg.sender;
        uint128 fee = _getFee(sale.voucher, amount_);
        uint128 price = PriceManager.price(sale.priceType, sale.saleId);
        uint256 units256;
        if (markets[sale.voucher].feePayType == FeePayType.BUYER_PAY) {

            //@audit the Principal = (amount_ - fee)
            units256 = amount_
                .sub(fee, "fee exceeds amount")
                .mul(uint256(markets[sale.voucher].precision))
                .div(uint256(price));
        } else {
            ......
        }

       ......

        _buy(buyer, sale, amount_, units_, price, fee);
        return units_;
    }

function _buy(
        address buyer_,
        Sale storage sale_,
        uint256 amount_,
        uint128 units_,
        uint128 price_,
        uint128 fee_
    ) internal {

        ......

        vars.feePayType = markets[sale_.voucher].feePayType;

        if (vars.feePayType == FeePayType.BUYER_PAY) {

            //@audit This means that the buyer needs to transfer the
            //       amount_ + fee amount of funds
            vars.transferInAmount = amount_.add(fee_);
            vars.transferOutAmount = amount_;
        } else if (vars.feePayType == FeePayType.SELLER_PAY) {

            ......
        }

        ERC20TransferHelper.doTransferIn(
            sale_.currency,
            buyer_,
            vars.transferInAmount
        );

        ......
    }
```

**Suggestion**

We can avoid this problem by using the `amount` parameter when calculating the buyer's actual `units` received. In this case, the buyer's parameter `amount_` input is the number of funds spent without taking into account the commission. If fees are taken into account, the actual amount of funds paid will be greater than the value of this parameter. The buyer has already transferred the ether into the contract when the `buyByAmount()` function is executed due to the special nature of the ether. So the actual amount of ether transferred by the buyer will be `amount_ + fee`.

**Status**

The developer team has modified this issue in 55b26c9.

### 4.3.2 There is a logical problem when the buyer does not transfer enough ether to buy the specified number of units.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Medium | Design logic | Fixed |

**Description**

A buyer can buy the specified number of `units` in a voucher by calling the `buyByUnits()` function. For buyers paying for an ERC20 token, the code will calculate the money needed to buy the specified number of `units`. If the buyer is required to pay the fee, this fee will be charged additionally.

When the buyer pays the type of funds as ether, the calculation of the fee in the code is different from the normal logic. Consider the following case: assuming that the buyer does not transfer enough ether to cover the number of `units` he wants to buy, the number of units the buyer can actually get will be recalculated based on the actual number of ether transferred by the buyer. However, ether needs to be transferred simultaneously when the buyer calls the `buyByUnits()` function. We can necessarily break down the money transferred by the buyer into two parts, namely.

$$msg.value \ = \ number\ of\ funds\ needed\ to\ buy\ units \ + \ platform\ fee$$

Therefore, the actual amount of funds used by users to purchase `units` (excluding the commission component) should be as follows:

$$funds\ used = (msg.value * FULL\_PERCENTAGE)/(FULL\_PERCENTAGE + feeRate)$$

Accordingly, the platform fees should be as follows:

$$Platform\ fee = funds\ used\ *\ feeRate\ /\ FULL\_PERCENTAGE$$

The current code takes `msg.value`, the total amount of ether transferred to the contract by the buyer, as the principal amount of `units` purchased by the user, and uses this data to calculate the number of fees received by the platform, which obviously does not take into account that the platform fees are already included in `msg.value`, which results in a logical error.

```
// @audit located in SolvConvertibleMarket.sol
function buyByUnits(uint24 saleId_, uint128 units_)
        external
        payable
        virtual
        override
        returns (uint256 amount_, uint128 fee_)
    {
        Sale storage sale = sales[saleId_];
        address buyer = msg.sender;
        uint128 price = PriceManager.price(sale.priceType, sale.saleId);

        amount_ = uint256(units_).mul(uint256(price)).div(
            uint256(markets[sale.voucher].precision)
        );

        if (
            sale.currency == Constants.ETH_ADDRESS &&
            sale.priceType == PriceType.DECLIINING_BY_TIME &&
            amount_ != msg.value
        ) {
            amount_ = msg.value;

            uint128 fee = _getFee(sale.voucher, amount_);
            uint256 units256;
            if (markets[sale.voucher].feePayType == FeePayType.BUYER_PAY) {

                //@audit There is a problem with
                //       the logic of calculating the buyer's principal
                units256 = amount_
                    .sub(fee, "fee exceeds amount")
                    .mul(uint256(markets[sale.voucher].precision))
                    .div(uint256(price));
            } else {
                units256 = amount_
                    .mul(uint256(markets[sale.voucher].precision))
                    .div(uint256(price));
            }
            require(units256 <= uint128(-1), "exceeds uint128 max");
            units_ = uint128(units256);
        }

        fee_ = _getFee(sale.voucher, amount_);
        ......
    }
```

**Status**

The development team has modified this part of the code to disallow ether use if the commission is paid by the buyer in commit 55b26c9.

### 4.3.3 There is a logical problem with the way the `buyByUnits()` function calculates the commission when the buyer transfers a quantity of ether exactly equal to the value of the quantity of `units` he specifies.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Medium | Design logic | Fixed |

**Description**

A buyer can purchase a specified number of `units` in a voucher by calling the `buyByUnits()` function. For buyers paying for an ERC20 token, the code will calculate the amount of money needed to buy the specified number of `units`. If the buyer is required to pay a fee, this fee will be charged additionally.

When a buyer pays an ether, the commission in the code is calculated differently from the normal logic. Consider the following case: suppose that the amount of ether transferred by the buyer at this time, `msg.value`, is exactly equal to the amount of money the buyer would have spent without the fee, `amount_`. We have:

$$msg.value = amount\_$$

Suppose the buyer pays the platform fee. In that case, the `doTransferIn()` function must be executed to determine the buyer's amount of ether. We have：

$$msg.value = vars.transferInAmount$$

and at this point, the value of the parameter `vars.transferInAmount` is：

$$vars.transferInAmount = amount\_ + fee$$

It can be deduced that：

$$msg.value = amount\_ = amount\_ + fee$$

Obviously, in the case of the commission `fee != 0`, the function call will fail.

```
// @audit located in SolvConvertibleMarket.sol
function buyByUnits(uint24 saleId_, uint128 units_)
        external
        payable
        virtual
        override
        returns (uint256 amount_, uint128 fee_)
    {
        Sale storage sale = sales[saleId_];
        address buyer = msg.sender;
        uint128 price = PriceManager.price(sale.priceType, sale.saleId);


        amount_ = uint256(units_).mul(uint256(price)).div(
            uint256(markets[sale.voucher].precision)
        );

        //@audit assume that amount_ == msg.value
        ......

        fee_ = _getFee(sale.voucher, amount_);

        uint256 err = solver.operationAllowed(
            "buyByUnits",
            abi.encode(
                sale.voucher,
                sale.tokenId,
                saleId_,
                buyer,
                sale.currency,
                amount_,
                units_,
                price
            )
        );
        require(err == 0, "solver not allowed");

        _buy(buyer, sale, amount_, units_, price, fee_);
        return (amount_, fee_);
    }

function _buy(
        address buyer_,
        Sale storage sale_,
        uint256 amount_,
        uint128 units_,
        uint128 price_,
        uint128 fee_
    ) internal {
        ......

        sale_.units = sale_.units.sub(units_, "insufficient units for sale");
        BuyLocalVar memory vars;
        vars.feePayType = markets[sale_.voucher].feePayType;

        if (vars.feePayType == FeePayType.BUYER_PAY) {
            vars.transferInAmount = amount_.add(fee_);
            vars.transferOutAmount = amount_;
        } else if (vars.feePayType == FeePayType.SELLER_PAY) {
            ......
        }

        ERC20TransferHelper.doTransferIn(
            sale_.currency,
            buyer_,
            // @audit the argument equals amount_ + fee
            vars.transferInAmount
        );

        ......
    }

//@auidt located in ERC20TransferHelper.sol
function doTransferIn(
        address underlying,
        address from,
        uint256 amount
    ) internal returns (uint256) {
        if (underlying == Constants.ETH_ADDRESS) {
            // Sanity checks
            require(tx.origin == from || msg.sender == from, "sender
mismatch");
            //@auidt amount == vars.transferInAmount
            require(msg.value == amount, "value mismatch");

            return amount;
        } else {

            ......
    }
```

### Status

The development team has modified this part of the code to disallow ether use if the commission is paid by the buyer in commit 55b26c9.

### 4.3.4 The `_mintVoucher()` function needs to adjust the corresponding authorization amount so that the transfer amount matches the authorization amount.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Info | Design logic | Fixed |

**Description**

The `mintVoucher()` function locks the assets purchased by the purchaser into the VNFT token. The internal function `approve()` in the code authorizes the `voucherPool` contract to transfer the `asset` assets under the current contract. The number of assets transferred is `parameter.tokenInAmount`. It indicates the total number of assets sold by the issuer. In practice, it is common for purchasers not to buy all of the issuer's assets for sale at once but to buy different amounts of assets for sale multiple times. The actual number of assets purchased by the purchaser is `tokenInAmount`, which is typically less than `parameter.tokenInAmount`. To avoid potential security risks, the code should match the number of authorized assets authorized with the actual number transferred.

```
// @audit located in InitialConvertibleOfferingMarket.sol
function _mintVoucher(uint24 offeringId_, uint128 units_)
        internal
        virtual
        override
        returns (uint256 voucherId)
    {
        Offering memory offering = offerings[offeringId_];
        MintParameter memory parameter = _mintParameters[offeringId_];

        //@audit This allowance will increase with each user purchase
        IERC20(markets[offering.voucher].asset).approve(
            markets[offering.voucher].voucherPool,
            parameter.tokenInAmount
        );

        //@audit actual number of assets transferred
        uint128 tokenInAmount = units_.div(parameter.lowestPrice);
        (, voucherId) = IConvertibleVoucher(offering.voucher).mint(
            offering.issuer,
            offering.currency,
            parameter.lowestPrice,
            parameter.highestPrice,
            parameter.effectiveTime,
            parameter.maturity,
            tokenInAmount
        );
    }
```

**Suggestion**

Adjust the amount of the authorization with reference to the modifications as follows:

```
function _mintVoucher(uint24 offeringId_, uint128 units_)
        internal
        virtual
        override
        returns (uint256 voucherId)
    {
        Offering memory offering = offerings[offeringId_];
        MintParameter memory parameter = _mintParameters[offeringId_];

        uint128 tokenInAmount = units_.div(parameter.lowestPrice);


        IERC20(markets[offering.voucher].asset).approve(
            markets[offering.voucher].voucherPool,
            tokenInAmount
        );

        (, voucherId) = IConvertibleVoucher(offering.voucher).mint(
            offering.issuer,
            offering.currency,
            parameter.lowestPrice,
            parameter.highestPrice,
            parameter.effectiveTime,
            parameter.maturity,
            tokenInAmount
        );
    }
```

**Status**

The developer team took our suggestions and modified the code in commit 1c88dab.

### 4.3.5 The function `mint()` without permission protection may have unexpected consequences.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Info | Design logic | Discussed |

**Description**

The `mint()` function implements the issuer asset lock. This function is external, allowing any user to call it directly. All parameters of the function are user-definable, which may have the effect of not conforming to the project design rules.

Consider the following scenario: a user calls the `mint()` function directly and falsifies information about the issuer, and sets the maximum number of tokens to be repaid with the parameter `tokenInAmount_ = 0`, then the user gets a voucher with `units == 0`. The normal code design logic would be to burn the voucher when the `units` of the voucher is zero. So it needs to be made clear whether any user is allowed to call the `mint()` function and mint a non-conforming voucher.

```
//@audit located in ConvertibleVoucher.sol
function mint(
        address issuer_,
        address fundCurrency_,
        uint128 lowestPrice_,
        uint128 highestPrice_,
        uint64 effectiveTime_,
        uint64 maturity_,
        uint256 tokenInAmount_
    )
        external
        override
        returns (uint256 slot, uint256 tokenId)
    {
        ......

        slot = getSlot(
            issuer_, fundCurrency_, lowestPrice_, highestPrice_,
            effectiveTime_, maturity_, 0
        );
        if (!getSlotDetail(slot).isValid) {
            convertiblePool.createSlot(
                issuer_, fundCurrency_, lowestPrice_, highestPrice_,
                effectiveTime_, maturity_, 0
            );
        }

        uint256 units = convertiblePool.mintWithUnderlyingToken(_msgSender(),
slot, tokenInAmount_);
        tokenId = VoucherCore._mint(_msgSender(), slot, units);

        solver.operationVerify(
            "mint",
            abi.encode(_msgSender(), issuer_, slot, tokenId, units)
        );
    }
```

**Status**

This issue has been discussed, and it is clear that there will be no impact on the protocol business logic.

### 4.3.6 Discussion of the logic of the `split()` code.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Info | Design logic | Discussed |

**Description**

The `split()` function allows the user to split the voucher they hold into vouchers with different `units`, which still have the same slot. The `split()` function does not restrict the range of values of the `units` array elements when the user splits. Splitting as many vouchers with `units == 0` is possible as desired. It is important to clarify whether this is the case as intended.

```
//@audit located in VoucherCore.sol
function split(uint256 tokenId_, uint256[] calldata splitUnits_)
        public
        virtual
        override
        returns (uint256[] memory newTokenIds)
    {
        require(splitUnits_.length > 0, "empty splitUnits");
        newTokenIds = new uint256[](splitUnits_.length);

        for (uint256 i = 0; i < splitUnits_.length; i++) {
            uint256 newTokenId = _generateTokenId();
            newTokenIds[i] = newTokenId;
            VNFTCoreV2._split(tokenId_, newTokenId, splitUnits_[i]);
            voucherSlotMapping[newTokenId] = voucherSlotMapping[tokenId_];
        }
    }
//@audit located in VocherCoreV2.sol
function _split(
        uint256 tokenId_,
        uint256 newTokenId_,
        uint256 splitUnits_
    ) internal virtual {
        require(
            _isApprovedOrOwner(_msgSender(), tokenId_),
            "VNFT: not owner nor approved"
        );
        require(!_exists(newTokenId_), "new token already exists");

        _units[tokenId_] = _units[tokenId_].sub(splitUnits_);

        address owner = ownerOf(tokenId_);
        _mintUnits(owner, newTokenId_, _slotOf(tokenId_), splitUnits_);

        emit Split(owner, tokenId_, newTokenId_, splitUnits_);
    }
```

**Status**

This issue has been discussed, and it was confirmed that there is no impact on the protocol business logic.

### 4.3.7 Adds the `_afterTransferUnits()` function to improve the capability of the `_transferUnitsFrom()` function.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Info | Design logic | Discussed |

**Description**

The `_transferUnitsFrom()` function enables the transfer of `units` between different vouchers. The vouchers on the sending and receiving sides must have the same slot. To cope with complex business requirements, the `_transferUnitsFrom()` function has a built-in `_beforeTransferUnits()` function. This built-in function is a pre-built interface that allows the user to design specific business logic based on the actual usage scenario. To improve the functionality of the `_transferUnitsFrom()` function, it is recommended that the `_afterTransferUnits()` function be added after the `units` transfer is completed.

```
//@audit located in VNFTCoreV2.sol
function _transferUnitsFrom(
        address from_,
        address to_,
        uint256 tokenId_,
        uint256 targetTokenId_,
        uint256 transferUnits_
    ) internal virtual {
        require(from_ == ownerOf(tokenId_), "source token owner mismatch");
        require(to_ != address(0), "transfer to the zero address");

        _beforeTransferUnits(
            from_,
            to_,
            tokenId_,
            targetTokenId_,
            transferUnits_
        );

       ......

        emit TransferUnits(
            from_,
            to_,
            tokenId_,
            targetTokenId_,
            transferUnits_
        );
    }
```

**Status**

This issue has been discussed, and the development team has decided to revise it in a subsequent version.

## 5. Conclusion

After auditing and analyzing the Solv protocol contract, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above. SECBIT Labs holds the view that Solv protocol has good code quality, concise implementation, and detailed documentation.

## Disclaimer

# APPENDIX

## Vulnerability/Risk Level Classification

| Level | Description |
|---|---|
| High | Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock ethers 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. |