SALUS SECURITY

SEP 2024

# CODE SECURITY ASSESSMENT

XUNION

# Overview

## Project Summary

- Name: Xunion - xslc
- Platform: EVM-compatible chains
- Language: Solidity
- Repository:
  - https://github.com/artixv/xslc
- Audit Range: See Appendix - 1

# Project Dashboard

## Application Summary

| | |
|---|---|
| **Name** | Xunion - xslc |
| **Version** | v4 |
| **Type** | Solidity |
| **Dates** | Sep 12 2024 |
| **Logs** | Aug 16 2024; Aug 30 2024; Sep 11 2024; Sep 12 2024 |

## Vulnerability Summary

| | |
|---|---|
| **Total High-Severity issues** | 2 |
| **Total Medium-Severity issues** | 5 |
| **Total Low-Severity issues** | 4 |
| **Total informational issues** | 2 |
| **Total** | 13 |

## Contact

E-mail: support@salusec.io

# Risk Level Description

| | |
|---|---|
| High Risk | The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users. |
| Medium Risk | The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact. |
| Low Risk | The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances. |
| Informational | The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth. |

# Content

# Introduction

## 1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

## 1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

## 1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

# Findings

## 2.1 Summary of Findings

| ID | Title | Severity | Category | Status |
|---|---|---|---|---|
| 1 | Decimal inconsistencies cause users being able to lend more money | High | Numerics | Resolved |
| 2 | When users return $SLC, the riskIsolationModeAmount is not reduced | High | Business logic | Resolved |
| 3 | Native token may be locked | Medium | Business Logic | Resolved |
| 4 | Incorrect calculation of the liquidation amount | Medium | Business Logic | Resolved |
| 5 | Incomplete validation leads to miscalculation of health factor | Medium | Business Logic | Resolved |
| 6 | Users can execute containSLC() multiple times in the same block and block other user | Medium | Business Logic | Resolved |
| 7 | Centralization risk | Medium | Centralization | Acknowledged |
| 8 | Loss of precision | Low | Numerics | Resolved |
| 9 | Missing checks for maximumLTV and liquidationPenalty | Low | Data Validation | Resolved |
| 10 | Missing check to verify if the asset is licensed | Low | Data Validation | Resolved |
| 11 | Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom() | Low | Data Validation | Resolved |
| 12 | Incorrect error message | Informational | Inconsistency | Resolved |
| 13 | Use of floating pragma | Informational | Configuration | Resolved |

## 2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

| 1. Decimal inconsistencies cause users being able to lend more money | |
|---|---|
| Severity: High | Category: Numerics |
| Target:<br>- contracts/slcvaults.sol | |

### Description

The protocol does not normalize decimals across different assets.

This may cause users to receive less $SLC than expected when using tokens with fewer than 18 decimals as collateral. Similarly, when tokens with fewer than 18 decimals are used as the mainCollateralToken, calling the slcTokenBuy() function to purchase $SLC will yield less $SLC than expected.

contracts/slcvaults.sol:L168-L178

```
function viewUsersHealthFactor(address user) public view returns(uint userHealthFactor,
uint userAssetsValue, uint userBorrowedSLCAmount, uint userAvailbleBorrowedSLCAmount){
   ...
    for(uint i=0;i<assetsSerialNumber.length;i++){
        if(licensedAssets[assetsSerialNumber[i]].maxDepositAmount == 0){
            tempValue[0] += userAssetsMortgageAmount[user][assetsSerialNumber[i]] *
iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1 ether;
            tempLoanToValue[0] += userAssetsMortgageAmount[user][assetsSerialNumber[i]]
* iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1 ether
                                * licensedAssets[assetsSerialNumber[i]].maximumLTV /
10000;
        }else if(userModeAssetsAddress[user]==assetsSerialNumber[i]){
            tempValue[1] += userAssetsMortgageAmount[user][assetsSerialNumber[i]] *
iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1 ether;
            tempLoanToValue[1] += userAssetsMortgageAmount[user][assetsSerialNumber[i]]
* iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1 ether
                                * licensedAssets[assetsSerialNumber[i]].maximumLTV /
10000;
        }
    }
    ...
}
```

contracts/slcvaults.sol:L290

```
function slcTokenBuy(address TokenAddr, uint amount) public returns(uint outputAmount){
    ...
    if(tokens[0] == tokens[2]){
        outputAmount = amount * 1 ether * 99 / (100 * slcValue);
    }
    ...
}
```

## Recommendation

It is recommended to normalize the decimals.

## Status

The team has resolved this issue in commit a4455552, 7ccc7cb.

| 2. When users return $SLC, the riskIsolationModeAmount is not reduced | |
|---|---|
| Severity: High | Category: Business logic |
| Target:<br>- contracts/slcvaults.sol | |

## Description

When a user borrows `$SLC` through the `obtainSLC()` function, if the user's collateral is an isolated asset, the borrowed amount is added to the `riskIsolationModeAmount` variable. This amount is checked against a borrowing limit; once the limit is reached, the user cannot borrow any more. However, when the user repays the loan through the returnSLC() function or if their assets are liquidated, the corresponding amount is not deducted from this variable.

As a result, even after repaying the loan, the user will not be able to borrow `$SLC` again if the borrowing limit has been reached.

contracts/slcvaults.sol:L383-L386

```
function obtainSLC(uint amount, address user) public  {
    ...
    if(userMode[user] == 1){
        riskIsolationModeAmount[userModeAssetsAddress[user]] += amount;
        require(riskIsolationModeAmount[userModeAssetsAddress[user]] <=
licensedAssets[userModeAssetsAddress[user]].maxDepositAmount,"Amount Exceed Limit");
    }
    ...

}
```

## Recommendation

Consider subtracting the corresponding amount from the `riskIsolationModeAmount` variable when the user repays the loan or the user's assets are liquidated.

## Status

The team has resolved this issue in commit [a1f89ff](a1f89ff).

| 3. Native token may be locked | |
|---|---|
| Severity: Medium | Category: Business logic |
| Target:<br>- contracts/slcoracle.sol | |

## Description

contracts/slcoracle.sol:L82-L85

```
function pythPriceUpdate(bytes[] calldata updateData) public payable {
    uint fee = IPyth(pythAddr).getUpdateFee( updateData);
    IPyth(pythAddr).updatePriceFeeds{ value: fee }(updateData);
}
```

When the `pythPriceUpdate()` function is called to update the price, the caller must pay a fee. This fee is intended for use when the updatePriceFeeds() function is called externally. However, the amount used may differ from the amount paid, and any excess native tokens are not refunded. Additionally, the contract lacks functions to manage native tokens, which could lead to excess tokens being locked within the contract.

## Recommendation

Consider returning any remaining native tokens after the function execution is complete.

## Status

The team has resolved this issue in commit a1f89ff.

| 4. Incorrect calculation of the liquidation amount | |
|---|---|
| Severity: Medium | Category: Business logic |
| Target:<br>- contracts/slcvaults.sol | |

## Description

The `tokenLiquidateEstimate()` function is used to estimate the maximum amount of token that can be liquidated when the user's health factor is insufficient.

In the `tokenLiquidate()` function, the calculation for the amount of `$SLC` required to liquidate collateral is done using the formula `amount * 1 ether * (10000 - licensedAssets[token].liquidationPenalty) / (10000 * slcValue)`.

However, this same formula is used in the tokenLiquidateEstimate() function to estimate the collateral amount that can be liquidated with a given amount of $SLC. This may cause the estimated maximum liquidatable amount to be smaller than the actual value.

contracts/slcvaults.sol:L437-L542

```
function tokenLiquidateEstimate(address user,address token) public view returns(uint
maxAmount){
    ...
    if(tokens[0] == mainCollateralToken){
        maxAmount = tempAmount * 1 ether * (10000 -
licensedAssets[token].liquidationPenalty) / (10000 * slcValue);
    }else{
        (maxAmount,) = ixInterface(xInterface).xExchangeEstimateOutput(tokens,
tempAmount);
        maxAmount = maxAmount * 1 ether * (10000 -
licensedAssets[token].liquidationPenalty) / (10000 * slcValue);
    }
    ...

}
```

## Recommendation

Consider correcting the calculation logic in the `tokenLiquidateEstimate()` function.

```
function tokenLiquidateEstimate(address user,address token) public view returns(uint
maxAmount){
    ...
    if(tokens[0] == mainCollateralToken){
        maxAmount = tempAmount * 1 ether * (10000 * slcValue) / (10000 -
licensedAssets[token].liquidationPenalty);
    }else{
        (maxAmount,) = ixInterface(xInterface).xExchangeEstimateOutput(tokens,
tempAmount);
        maxAmount = maxAmount * 1 ether * (10000 * slcValue) / (10000 -
```

```
licensedAssets[token].liquidationPenalty);
    }
    ...

}
```

## Status

The team has resolved this issue in commit a445552.

| 5. Incomplete validation leads to miscalculation of health factor | |
|---|---|
| Severity: Medium | Category: Business logic |
| Target:<br>- contracts/slcvaults.sol | |

## Description

The `usersHealthFactorEstimate()` function is used to calculate the expected impact on a user's health factor after performing a specific action.

contracts/slcvaults.sol:L411-L469

```
function usersHealthFactorEstimate(address user,address token,uint amount,bool operator)
public view returns(uint userHealthFactor){
    ...

    if(userObtainedSLCAmount[user] > 0){
        ...
    }else{
        userHealthFactor = 1000 ether;
    }
}
```

However, the function does not correctly return the change in health factor after minting SLC when `userObtainedSLCAmount[user] = 0`.

### Attach Scenario

1. If the total value of the user's collateral is $100

2. userObtainedSLCAmount[user] = 0

3. The user wants to mint $50 worth of SLC,so tempValue = 50, and operator is false

4. The expected userHealthFactor should be 100 / 50 * 1 ether = 2 ether, not 1000 ether

## Recommendation

Consider changing the above highlighted statement to:

```
if(userObtainedSLCAmount[user] > 0 || (userObtainedSLCAmount[user]== 0 && !operator))
    ...
}else{
    userHealthFactor = 1000 ether;
}
```

## Status

The team has resolved this issue in commit a1f89ff.

## 6. Users can execute containSLC() multiple times in the same block and block other user

| | |
|---|---|
| Severity: Medium | Category: Business logic |
| Target:<br>- contracts/slcvaults.sol<br>- contracts/slcinterface.sol | |

## Description

contracts/slcinterface.sol:L184-L189

```
function obtainSLC(uint amount) public {
    require(UserLatestBlockNumber[msg.sender] < block.number,"SLC Interface: Same block can only have ONE obtain operation");
    UserLatestBlockNumber[msg.sender] = block.number;
    iSlcVaults(slcVaults).obtainSLC(amount, msg.sender);
    IERC20(superLibraCoin).transfer(msg.sender,amount);
}
```

contracts/slcvaults.sol:L372-L393

```
function obtainSLC(uint amount, address user) public  {
    if(slcInterface[msg.sender]==false){
        require(user == msg.sender,"SLC Vaults: Not registered as slcInterface or user need be msg.sender!");
        require(latestBlockNumber < block.number,"SLC Vaults: Same block can only have ONE obtain operation ");
    }
    latestBlockNumber = block.number;
    ...

}
```

The `obtainSLC()` function controls that a user can only call obtainSLC once per block, but this functionality has been incorrectly implemented.

If a user calls `obtainSLC()` directly through the slcVaults contract, only the first caller in each block can succeed. Subsequent calls will fail due to `latestBlockNumber < block.number`. Moreover, the first user who successfully calls `obtainSLC()` can still invoke it again through the `slcinterface` contract, potentially executing `obtainSLC()` twice within the same block.

## Recommendation

Consider limiting `user` to operate only once in a block in `slcvaults` only.

## Status

The team has resolved this issue in commit [dba37ea](dba37ea).

| 7. Centralization risk | |
|---|---|
| Severity: Medium | Category: Centralization |
| Target:<br>- All | |

## Description

All contracts have a privileged account `setter`. `setter` has the authority to modify many critical parameters in the contract and to change other privileged addresses, such as `rebalancer` and `slcManager`.

If `setter`'s private key is compromised, an attacker can set themselves as the `slcManager`, allowing them to mint or burn `$SLC` at will.

contracts/superlibracoin.sol:L57-L73

```
function mintSLC(address _account,uint256 _value) public onlyManager lock{
    ...
    _mint(_account, _value);
}

function burnSLC(address _account,uint256 _value) public onlyManager lock{
    ...
    _burn(_account, _value);
}
```

They can also set themselves as the `rebalancer`, enabling them to withdraw any assets from the `slcVaults` contract, among other potential attacks.

contracts/slcvaults.sol:L488-L492

```
function excessAssetsReturn(address token, uint amount) public onlyRebalancer(){
    IERC20(token).transfer(msg.sender,amount);
    licensedAssets[token].mortgagedAmountReturned += amount;
    emit MortgagedAmountReturned(token, amount);
}
```

If the privileged accounts are plain EOA accounts, this can be worrisome and pose a risk to the other users.

## Recommendation

We recommend transferring privileged accounts to multi-sig accounts with timelock governors for enhanced security. This ensures that no single person has full control over the accounts and that any changes must be authorized by multiple parties.

## Status

This issue has been acknowledged by the team.

| 8. Loss of precision | |
|---|---|
| Severity: Low | Category: Numerics |
| Target:<br>- contracts/slcvaults.sol | |

## Description

contracts/slcvaults.sol:L163-L205

```
function viewUsersHealthFactor(address user) public view returns(uint userHealthFactor,
uint userAssetsValue, uint userBorrowedSLCAmount, uint userAvailbleBorrowedSLCAmount){
    ...
    for(uint i=0;i<assetsSerialNumber.length;i++){
        if(licensedAssets[assetsSerialNumber[i]].maxDepositAmount == 0){
            tempValue[0] += userAssetsMortgageAmount[user][assetsSerialNumber[i]] *
iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1 ether;
            tempLoanToValue[0] += userAssetsMortgageAmount[user][assetsSerialNumber[i]]
* iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1 ether
                                * licensedAssets[assetsSerialNumber[i]].maximumLTV /
10000;
        }else if(userModeAssetsAddress[user]==assetsSerialNumber[i]){
            tempValue[1] += userAssetsMortgageAmount[user][assetsSerialNumber[i]] *
iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1 ether;
            tempLoanToValue[1] += userAssetsMortgageAmount[user][assetsSerialNumber[i]]
* iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1 ether
                                * licensedAssets[assetsSerialNumber[i]].maximumLTV /
10000;
        }
    }

    userBorrowedSLCAmount = userObtainedSLCAmount[user];
    userAssetsValue = tempValue[0] + tempValue[1];

    if(userObtainedSLCAmount[user] > 0){
        if(userMode[user] == 0){
            userHealthFactor = (tempLoanToValue[0] * 1 ether /
userObtainedSLCAmount[user]) * slcValue / 1 ether;
            userAvailbleBorrowedSLCAmount = tempLoanToValue[0] * 1 ether / 1.2 ether;
        }else{
            userHealthFactor = (tempLoanToValue[1] * 1 ether /
userObtainedSLCAmount[user]) * slcValue / 1 ether;
            userAvailbleBorrowedSLCAmount = tempLoanToValue[1] * 1 ether / 1.2 ether;
        }
    }else{
        userHealthFactor = 1000 ether;
        if(userMode[user] == 0){
            userAvailbleBorrowedSLCAmount = tempLoanToValue[0] * 1 ether / 1.2 ether;
        }else{
            userAvailbleBorrowedSLCAmount = tempLoanToValue[1] * 1 ether / 1.2 ether;
        }
    }
    ...
}
```

In the `viewUsersHealthFactor()` function, the calculation of `userHealthFactor` uses the formula userAssetsMortgageAmount * price / 1 ether * maximumLTV / 10000 * 1 ether / 1.2 ether.

However, due to precision loss during division, this approach results in the actual userHealthFactor being lower than expected. Mathematically, the 1 ether terms can be canceled out, simplifying the calculation.

## Recommendation

Consider removing the redundant operations that cause precision loss.

## Status

The team has resolved this issue in commit dba37ea.

## 9. Missing checks for maximumLTV and liquidationPenalty

| Severity: Low | Category: Data Validation |
|---|---|
| Target:<br>- contracts/slcvaults.sol | |

## Description

The `maximumLTV` and `liquidationPenalty` variables have upper limits, but there are no checks in place to ensure that the assigned values do not exceed these limits.

contracts/slcvaults.sol:L134-L150

```
function licensedAssetsRegister(address _asset, uint MaxLTV, uint LiqPenalty,uint
MaxDepositAmount) public onlySetter {
    ...
    licensedAssets[_asset].maximumLTV = MaxLTV;
    licensedAssets[_asset].liquidationPenalty = LiqPenalty;
    ...
}
function licensedAssetsReset(address _asset, uint MaxLTV, uint LiqPenalty,uint
MaxDepositAmount) public onlySetter {
    ...
    licensedAssets[_asset].maximumLTV = MaxLTV;
    licensedAssets[_asset].liquidationPenalty = LiqPenalty;
    ...
}
```

contracts/slcvaults.sol:L34-L42

```
struct licensedAsset{
    ...
    uint     maximumLTV;           // for a home against the value of that property.(MAX
= 10000)
    uint     liquidationPenalty;   // MAX = 10000 ,default is 500(5%)
    ...
}
```

## Recommendation

Consider adding checks to ensure that the assigned values for `maximumLTV` and `liquidationPenalty` do not exceed the specified upper limits.

## Status

The team has resolved this issue in commit dba37ea.

| 10. Missing check to verify if the asset is licensed | |
|---|---|
| Severity: Low | Category: Data Validation |
| Target:<br>- contracts/slcvaults.sol | |

## Description

When a user calls the `licensedAssetsPledge()` function to deposit collateral, the function does not check whether the asset is licensed. This may lead to users depositing unauthorized collateral, which can prevent them from borrowing.

## Recommendation

Consider adding a check to ensure that the deposited asset is licensed.

## Status

The team has resolved this issue in commit a1f89ff.

| 11. Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom() | |
|---|---|
| Severity: Low | Category: Risky External Calls |
| Target:<br>- contracts/slcinterface.sol<br>- contracts/slcvaults.sol | |

## Description

Tokens not compliant with the ERC20 specification could return false from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

```
Callers MUST handle false from returns (bool success). Callers MUST NOT assume that
false is never returned!
```

## Recommendation

Consider using the `SafeERC20` library implementation from OpenZeppelin and call safeTransfer or safeTransferFrom when transferring ERC20 tokens.

## Status

The team has resolved this issue in commit dba37ea.

# 2.3 Informational Findings

| 12. Incorrect error message | |
|---|---|
| Severity: Informational | Category: Inconsistency |
| Target:<br>- contracts/slcvaults.sol | |

## Description

contracts/slcvaults.sol:L366

```
function redeemPledgedAssets(address TokenAddr, uint amount, address user) public  {
    ...
    require( factor >= 1.2 ether,"Your Health Factor <= 1.2, Cant redeem assets");
    ...
}
```

contracts/slcvaults.sol:L497

```
function tokenLiquidate(address user,address token, uint amount) public returns(uint
outputAmount) {
    require(amount > 0,"SLC Vaults: Cant Pledge 0 amount");
    ...
}
```

The code above does not match the description of the error message.

## Recommendation

Consider correcting the code or the error message.

## Status

The team has resolved this issue in commit [a1f89ff](a1f89ff).

| 13. Use of floating pragma | |
|---|---|
| Severity: Informational | Category: Configuration |
| Target:<br>- All | |

## Description

```
pragma solidity ^0.8.0;
```

All contracts use a floating compiler version ^0.8.10.

Using a floating pragma ^0.8.10 statement is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.

## Recommendation

It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

## Status

The team has resolved this issue in commit dba37ea.

# Appendix

## Appendix 1 - Files in Scope

This audit covered the following files in commit 4835938:

| File | SHA-1 hash |
|---|---|
| contracts/slcinterface.sol | 2fdfdfaefe76dbc2ce893dcc7494a0a2e6af080b |
| contracts/slcoracle.sol | 2cfc1112545542d78034d17a8f7cc395b746225c |
| contracts/slcvaults.sol | 3a77cadeb2679ac4852d94f7f6e7255e8bc077f8 |
| contracts/superlibracoin.sol | 46d890ac05c5ac553f9159af7fc45f8323c208dd |