SALUS SECURITY

SEP 2024

CODE SECURITY ASSESSMENT

ARTIXV

Overview

Project Summary

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

Project Dashboard

Application Summary

Name	Artixv - xLending
Version	v3
Type	Solidity
Dates	Sep 11 2024
Logs	Jul 29 2024; Aug 30 2024; Sep 10 2024

Vulnerability Summary

Total High-Severity issues	3
Total Medium-Severity issues	2
Total Low-Severity issues	4
Total informational issues	4
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	The lack of permission control on createDeAndLoCoin() causes the protocol to fail to add new assets	High	Access Control	Resolved
2	Decimal inconsistencies cause users being able to lend more money	High	Numerics	Resolved
3	Asset status not updated during liquidation	High	Business logic	Resolved
4	Incorrect calculation of maxAmounts	Medium	Business logic	Resolved
5	Missing implementation	Medium	Code Quality	Resolved
6	Mismatch between code and comments	Low	Inconsistency	Resolved
7	Missing condition check	Low	Data Validation	Resolved
8	Loss of precision	Low	Numerics	Resolved
9	Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom()	Low	Risky External Calls	Resolved
10	ETH may be locked	Informational	Business logic	Resolved
11	Use of floating pragma	Informational	Configuration	Resolved
12	Redundant code	Informational	Redundancy	Resolved
13	Gas Optimization	Informational	Gas Optimization	Resolved

2.2 Notable Findings

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

1. The lack of permission control on createDeAndLoCoin() causes the protocol to fail to add new assets

Severity: High	Category: Access Control

Target:
- contracts/coinFactory.sol
- contracts/lendingManager.sol

Description

contracts/lendingManager.sol:L187

```
function licensedAssetsRegister(address _asset, ...) public onlySetter {
    ...
    licensedAssets[_asset].assetAddr = _asset;
    ...
    assetsDepositAndLend[_asset] = iCoinFactory(coinFactory).createDeAndLoCoin(_asset);
}
```

In the `lendingManager.sol`, when adding a new asset, the `coinFactory.createDeAndLoCoin()` function is called to create the corresponding depositToken and loanToken.

```
function createDeAndLoCoin(address token) external returns (address[2] memory
_pAndLCoin) {
    require(getDepositCoin[token] == address(0), 'Coin Factory: COIN_EXISTS');
     ...
    //Only ERC20 Tokens Can creat pairs
    _pAndLCoin[0] = address(new depositOrLoanCoin{salt: _salt1}(0,token,lendingManager,
rewardContract, strConcat(string(ERC20(token).symbol()), " Deposit
Coin"),strConcat(string(ERC20(token).symbol()), " DCoin")));  //
    _pAndLCoin[1] = address(new depositOrLoanCoin{salt: _salt2}(1,token,lendingManager,
rewardContract,strConcat(string(ERC20(token).symbol()), " Loan
Coin"),strConcat(string(ERC20(token).symbol()), " LCoin")));
    getDepositCoin[token] = _pAndLCoin[0];
    getLoanCoin[token] = _pAndLCoin[1];
    ...
}
```

However, the `createDeAndLoCoin()` function lacks proper access control. A malicious attacker could front-run this function to create a `depositOrLoanCoin` contract, which would cause the `lendingManager` contract to fail when registering this asset with `licensedAssetsRegister()` function.

Recommendation

Consider adding appropriate access control to the `createDeAndLoCoin()` function.

Status

The team has resolved this issue in commit 750ddc1.

2. Decimal inconsistencies cause users being able to lend more money

Severity: High	Category: Numerics
Target: - contracts/lendingManager.sol	

Description

When a user lends, the `userDepositAndLendingValue()` function calculates the total value of deposits and loans to check the health factor. However, this function does not normalize the decimals of different tokens, which can lead to inaccurate values when lending.

For example, users may be able to lend more USDC if they use ETH as collateral to lend USDC.

contracts/lendingManager.sol:L294-L327

```
function userDepositAndLendingValue(address user) public view returns(uint
_amountDeposit,uint _amountLending){
    require(assetsSerialNumber.length < 100,"Lending Manager: Too Much assets");

    for(uint i=0;i<assetsSerialNumber.length;i++){
        if(userMode[user]==1 && assetsSerialNumber[i] == userRIMAssetsAddress[user]){
            _amountDeposit  =
iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]][0]).balanceOf(user)
                            * iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1
ether
                            * licensedAssets[assetsSerialNumber[i]].maximumLTV /
UPPER_SYSTEM_LIMIT;
            _amountLending  =
iDepositOrLoanCoin(assetsDepositAndLend[riskIsolationModeAcceptAssets][1]).balanceOf(use
r)
                            *
iSlcOracle(oracleAddr).getPrice(riskIsolationModeAcceptAssets) / 1 ether;
            break;
        }
        if(userMode[user]>1){
            if(licensedAssets[assetsSerialNumber[i]].lendingModeNum == userMode[user]){
                _amountDeposit +=
iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]][0]).balanceOf(user)
                                * iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i])
/ 1 ether
                                *
licensedAssets[assetsSerialNumber[i]].homogeneousModeLTV / UPPER_SYSTEM_LIMIT;
                _amountLending +=
iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]][1]).balanceOf(user)
                                * iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i])
/ 1 ether;
            }
            continue;
        }
        if(userMode[user]==0){
            if(licensedAssets[assetsSerialNumber[i]].maxLendingAmountInRIM > 0){
                continue;
            }
            _amountDeposit +=
```

```
iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]][0]).balanceOf(user)
                            * iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1
ether
                            * licensedAssets[assetsSerialNumber[i]].maximumLTV /
UPPER_SYSTEM_LIMIT;
            _amountLending +=
iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumber[i]][1]).balanceOf(user)
                            * iSlcOracle(oracleAddr).getPrice(assetsSerialNumber[i]) / 1
ether;
        }
    }
}
```

Recommendation

It is recommended to normalize the decimals.

Status

The team has resolved this issue in commit [48e680b](48e680b).

3. Asset status not updated during liquidation

Severity: High	Category: Business logic
Target: - contracts/lendingManager.sol	

Description

When the contract assets change, the interest rate changes caused by the change should be recorded.

contracts/lendingManager.sol:L525-L545

```
function tokenLiquidate(address user,
                        address liquidateToken,
                        uint    liquidateAmount,
                        address depositToken) public returns(uint usedAmount) {
    ...
    iLendingVaults(lendingVault).vaultsERC20Approve(liquidateToken, liquidateAmount);
    IERC20(depositToken).transferFrom(msg.sender, lendingVault, usedAmount);
    IERC20(liquidateToken).transferFrom(lendingVault, msg.sender, liquidateAmount);
    ...
}
```

However, during the liquidation process, the function does not update these corresponding values. This lack of updates may lead to inconsistencies and inaccurate financial modeling within the system.

Recommendation

It is recommended to update the assetInfos mapping separately before and after performing the liquidation.

Status

The team has resolved this issue in commit 23956fe.

4. Incorrect calculation of maxAmounts

Severity: Medium	Category: Business logic
Target: - contracts/lendingManager.sol	

Description

contracts/lendingManager.sol:L568-L571

```
function tokenLiquidateEstimate(address user,
                        address liquidateToken,
                        address depositToken) public view returns(uint[2] memory
maxAmounts){
    ...
    }else{
        maxAmounts[1] = amountDeposit;
        maxAmounts[0] = amountDeposit * 1 ether /
iSlcOracle(oracleAddr).getPrice(depositToken);
    }
}
```

This function is intended to provide an estimate of the maximum amounts of tokens that can be liquidated for a given user, considering both the liquidation token and the deposit token.

However, the current implementation mistakenly utilizes the price of the `depositToken` when determining the quantity of the `liquidateToken` that can be liquidated, leading to an inaccurate estimation.

Recommendation

It is recommended to correctly use liquidateToken price for calculations.

Status

The team has resolved this issue in commit 062ea71.

5. Missing function implementation

Severity: Medium	Category: Code Quality
Target: - contracts/lendingInterface.sol	

Description

contracts/lendingInterface.sol:L30-L32

```
function assetsLiqPenaltyInfo(address token) external view returns(uint liqPenalty){
    return iLendingManager(lendingManager).assetsLiqPenaltyInfo(token);
}
```

The `assetsLiqPenaltyInfo()` function defined within the `lendingInterface.sol`, aims to retrieve the liquidation penalty information for a specific token from the `lendingManager` contract.

However, it attempts to interface with a functionality (assetsLiqPenaltyInfo) that has not been implemented in the targeted lendingManager contract.

Recommendation

Consider properly implementing the `assetsLiqPenaltyInfo()` function in the `lendingManager` contract.

Status

The team has resolved this issue in commit 48e680b.

6. Mismatch between code and comments

Severity: Low	Category: Inconsistency
Target: - contracts/lendingManager.sol	

Description

contracts/lendingManager.sol:L50

```
// MAX = UPPER_SYSTEM_LIMIT ,default is 500(5%)
```

contracts/lendingManager.sol:L170

```
&& _liqPenalty <= UPPER_SYSTEM_LIMIT/5
```

The comment indicates that the maximum value for `liquidationPenalty` is `UPPER_SYSTEM_LIMIT`, but the function actually enforces a maximum value of `UPPER_SYSTEM_LIMIT / 5`.

Recommendation

Consider correcting either the code or the comment to ensure consistency.

Status

The team has resolved this issue in commit 1fbb7d3.

7. Missing condition check

Severity: Low	Category: Data Validation
Target: - contracts/lendingManager.sol	

Description

When an asset is registered for the first time, the `licensedAssetsRegister()` function checks critical values within a valid range using a `require()` statement. However, when the asset is later reset, the `licensedAssetsReset()` function does not perform this value check.

contracts/lendingManager.sol:L169-L157

```
function licensedAssetsRegister(address _asset,
                                ...
                                uint _bestDepositInterestRate) public onlySetter {
    require(   _maxLTV < UPPER_SYSTEM_LIMIT
            && _liqPenalty <= UPPER_SYSTEM_LIMIT/5
            && _bestLendingRatio < UPPER_SYSTEM_LIMIT
            && _homogeneousModeLTV >= _maxLTV
            && _homogeneousModeLTV < UPPER_SYSTEM_LIMIT
            && _bestDepositInterestRate > 0
            && _bestDepositInterestRate < UPPER_SYSTEM_LIMIT,"Lending Manager: Exceed
UPPER_SYSTEM_LIMIT");
    ...
}
```

Recommendation

Consider adding a value check in the `licensedAssetsReset()` function to ensure that critical values are validated.

Status

The team has resolved this issue in commit 48e680b.

8. Loss of precision

Severity: Low	Category: Numerics
Target: - contracts/lendingManager.sol	

Description

contracts/lendingManager.sol:L535-L537

```
function tokenLiquidate(...) public returns(uint usedAmount) {
    ...
    usedAmount = liquidateAmount * iSlcOracle(oracleAddr).getPrice(liquidateToken) / 1
ether;
    usedAmount = usedAmount * (UPPER_SYSTEM_LIMIT -
licensedAssets[liquidateToken].liquidationPenalty) * 1 ether /
                                (UPPER_SYSTEM_LIMIT *
iSlcOracle(oracleAddr).getPrice(depositToken));
    require( amountDeposit >= usedAmount,"Lending Manager: amountLending >=
liquidateAmount");
}
```

In the tokenLiquidate() function, the calculation of usedAmount involves the operation a / b * c * b.

Mathematically, the b(1 ether) can be canceled out, simplifying the calculation. Also, due to the loss of precision in the division of a / b, this approach results in the actual liquidation amount being less than expected.

Recommendation

Consider removing the redundant operations that cause precision loss.

Status

The team has resolved this issue in commit 6ae1e3a.

9. Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom()

Severity: Low	Category: Risky External Calls
Target: - contracts/lendingManager.sol - contracts/lendingInterface.sol - contracts/lendingVaults.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 6ae1e3a.

2.3 Informational Findings

10. ETH may be locked

Severity: Informational	Category: Business logic
Target: - contracts/lendingVaults.sol	

Description

contracts/lendingVaults.sol:L67-L68

```
fallback() external payable {}
receive() external payable {}
```

The contract has both `fallback()` and `receive()` functions, but it lacks the logic to handle ETH.
This means that once an ETH has been transferred into the contract, it will be permanently locked in the contract.

Recommendation

Consider adding functions to handle ETH.

Status

The team has resolved this issue in commit 6ae1e3a.

11. Use of floating pragma

Severity: Informational	Category: Configuration
Target: - All	

Description

```
pragma solidity ^0.8.0;
```

Using a floating pragma ^0.8.0 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 6479495.

12. Redundant code

Severity: Informational	Category: Redundancy
Target: - contracts/lendingManager.sol	

Description

Unused code should be removed before deploying the contract to mainnet. We have identified the following variables are not being utilized:

contracts/lendingManager.sol:L21,L23

```
uint public slcValue;
address public xInterface;
```

Recommendation

Consider removing the redundant code.

Status

The team has resolved this issue in commit 6ae1e3a.

13. Gas Optimization

Severity: Informational	Category: Gas Optimization
Target: - contracts/lendingManager.sol - contracts/lendingInterface.sol	

Description

In general, if a state variable is read more than once, caching its value to a local variable and reusing it will save gas since a storage read spends more gas than a memory write plus a memory read.
In the following loop, the storage variable ts.openTradeTokenIns.length is repeatedly accessed, leading to more gas usage.

contracts/lendingManager.sol:L281, L288, L297, L346, L375

```
for(uint i=0;i<assetsSerialNumber.length;i++)
```

contracts/lendingInterface.sol:L46

```
for(uint i=0;i<assetsSerialNumber.length;i++)
```

contracts/lendingInterface.sol:L122, L214, L225, L245, L280

```
for(uint i=0;i<tokens.length;i++){
```

Recommendation

Consider caching the array length in the stack to save gas.

Status

The team has partially resolved this issue in commit 6ae1e3a.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit a9e7bcd:

File	SHA-1 hash
contracts/coinFactory.sol	938eda46da759301d573cd57a0f14ca78c80e1d9
contracts/lendingCoreAlgorithm.sol	e2dda57f34a05a1c435ed01b531b36ace07d4499
contracts/lendingInterface.sol	38d0d426a654b790c66ba5c743e7445bed8c702c
contracts/lendingManager.sol	7fa068a803ac5d7ab7bc90589321215e8d4f2f8e
contracts/lendingVaults.sol	508f46687ebc5e5ff06ef10fbdca5db1c1cf55e8
contracts/template/depositOrLoanCoin.sol	b934a9ef572c5c948d5f91c669d9388f133b06a0