CREDITSWAP

SMART CONTRACTS REVIEW

zokyo

December 22nd 2023 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

SCORE

89

ZOKYO AUDIT SCORING CREDITSWAP

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 1 Critical issue: 1 resolved = 0 points deducted
- 2 High issues: 2 resolved = 0 points deducted
- 4 Medium issues: 3 resolved and 1 acknowledged = - 3 points deducted
- 8 Low issues: = 3 resolved and 5 acknowledged = - 5 points deducted
- 8 Informational issues: 2 resolved, 1 unresolved and 5 acknowledged = -3 points deducted

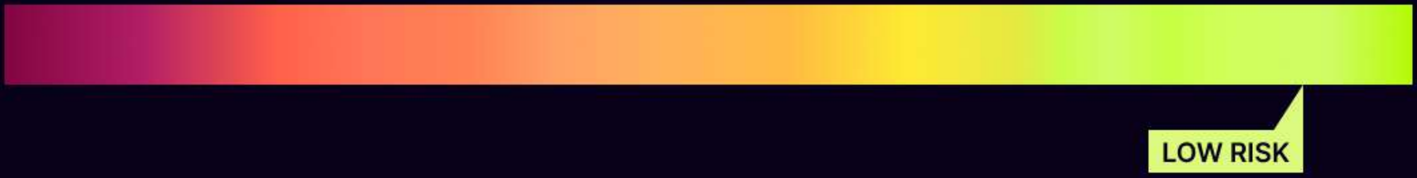
Thus, 100 - 3 - 5 - 3 = 89

TECHNICAL SUMMARY

This document outlines the overall security of the Creditswap smart contracts evaluated by the Zokyo Security team.

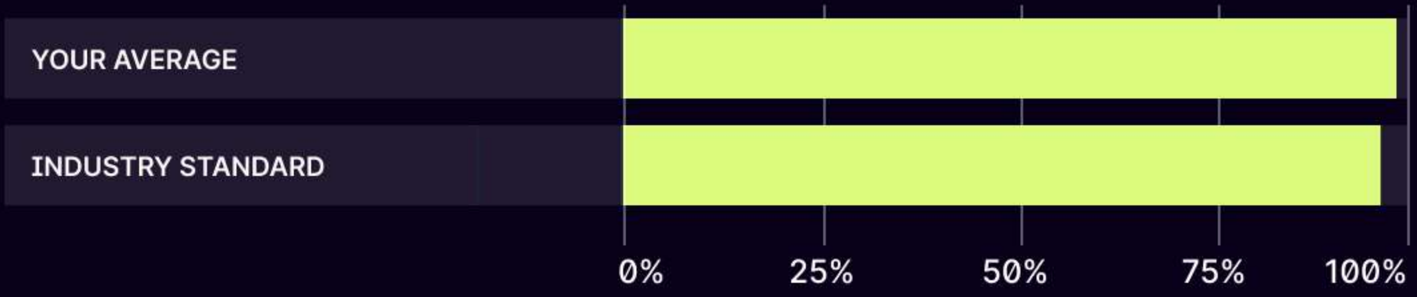
The scope of this audit was to analyze and document the Creditswap smart contracts codebase for quality, security, and correctness.

Contract Status



There was 1 critical issue found during the review. (See Complete Analysis)

Testable Code



97,22% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Creditswap put in place a bug bounty program to encourage further active analysis of the smart contracts.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Creditswap repository:
Repo: https://github.com/CreditSwap/core

Last commit - 156a56652dc87dc487bde6503c8f1b04b208735d

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./creditUsd/CreditUSD.sol
- ./creditUsd/CreditUSDMinter.sol
- ./util/DefaultCreditorAutomation.sol
- ./nft/DebtorNFT.sol
- ./nft/DebtorStaking.sol
- ./nft/CreditorNFT.sol
- ./nft/StakingPool.sol
- ./nft/NFTBase.sol
- ./oracle/BaseOracleUSD.sol
- ./oracle/AggregatedChainlinkOracle.sol
- ./oracle/WBTCOracle.sol
- ./LoanVault.sol
- ./ProtocolController.sol
- ./governance/token/VeCreditSwapToken.sol
- ./governance/token/CSProtocolToken.sol
- ./governance/token/lib/TokenWrapper.sol
- ./governance/token/lib/TokenCheckpointer.sol
- ./governance/RewardAuctionHouse.sol
- ./governance/RewardDistributor.sol
- ./governance/CSGovernor.sol
- ./governance/CSTimelock.sol
- ./lib/TimestampLib.sol
- ./lib/Beacon.sol
- ./lib/BeaconProxy.sol
- ./lib/AccessControl.sol
- ./lib/RenderNFT.sol
- ./lib/Staking.sol
- ./CreditSwapMarket.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most resent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Creditswap smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contracts by industry leaders.

03 Testing contracts logic against common and uncommon attack vectors.

04 Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team identified a critical severity issue and two high severity issues. Additionally, issues with medium and low severity levels, along with a couple of informational findings, were discovered. For a more detailed analysis of these findings, please refer to the "Complete Analysis" section.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Creditswap team and the Creditswap team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

● **Critical**

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

● **High**

The issue affects the ability of the contract to compile or operate in a significant way.

● **Medium**

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

● **Low**

The issue has minimal impact on the contract’s ability to operate.

● **Informational**

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Attacker Can Grief Liquidations And Repayments	Critical	Resolved
2	Attacker Can Grief mintWithSignature And mintWithAutomation Calls	High	Resolved
3	Incorrect Fee Accrual In Case Of A Referral	High	Resolved
4	No Price Staleness Check	Medium	Acknowledged
5	Unlimited Token Minting for CreditUSD and CSProtocolToken	Medium	Resolved
6	Reentrance in mint() of CreditUSDMinter	Medium	Resolved
7	Reentrance in CreditorNFT	Medium	Resolved
8	Liquidation In LounVault.sol should have a onlyCreditor Modifier	Low	Acknowledged
9	Liquidations Might Not Be Profitable	Low	Acknowledged
10	Ensure No Rounding	Low	Acknowledged
11	Missing zero address checks for CreditUSDMinter	Low	Resolved
12	Missing zero address and sanity checks in LoanVault	Low	Acknowledged
13	Missing zero address checks in ProtocolController	Low	Resolved
14	Missing zero address check in CreditUSD	Low	Acknowledged
15	Missing sanity checks in RewardAuctionHouse	Low	Resolved
16	Liquidations Might Revert If Collateral Is A Token Which Reverts On 0 Value Transfer	Informational	Resolved
17	USDC Blacklisting Impact on LoanVault Liquidation Process	Informational	Acknowledged

#	Title	Risk	Status
18	Asset Allowlist Check in LoanVault:	Informational	Acknowledged
19	Careful Usage of Upgradeable contracts	Informational	Acknowledged
20	Make Sure That Timelock Contract Is Deployed As Part Of The Deploy Script	Informational	Unresolved
21	Allowing Fee-On-Transfer Tokens Might Be Problematic	Informational	Acknowledged
22	Resolve TODOs	Informational	Acknowledged
23	CEI Violation	Informational	Resolved

CRITICAL-1 | RESOLVED

Attacker Can Grief Liquidations And Repayments

To liquidate an unhealthy loan position the liquidate() function inside CreditorNFT can be called by anyone where the debtAmount of debt token is paid out by the liquidator.
This function in turn calls the liquidate function of LoanVault at L133.

Inside LoanVault.sol's liquidate() it is checked if the debtAmount (initial debt amount when loan was created) is now equal to the balance of debt token in the vault , if not revert (L163)

An attacker can see a liquidation() call in the mempool and →

a.) Frontruns this call to send the lowest amount of debt token to the vault , say 1
b.) Now when the liquidator tries to liquidate he sends out debtAmount of tokens to the vault , let's say they were 100
c.) It is checked that debt amount and balance of debt token balance in the vault is equal
d.) But they are not since there are a total of 101 debt tokens now , liquidation reverts.

Due to this the vault/loan position can never be liquidated and the protocol will continue to incur huge losses as the collateral value falls down.

The same problem lies in repay functionality , at L150 in LoanVault.sol it will revert due to the same case as above and make it impossible for a debtor to repay their loan , resulting in forced liquidations.

Recommendation:

Have an internal accounting system or change the condition to if the balance in the vault is less than debt amount, then revert instead of a strict equality.

HIGH-1 | RESOLVED

Attacker Can Grief mintWithSignature And mintWithAutomation Calls

A debtor can accept a creditor’s loan position from the DebtorNFT.sol contract . Aside from the usual mint function (which accepts a loan offer an mints a debtor NFT) there are two more options , one is mintWithDebtorSignature which uses offchain signature and the other is mintWithAutomation which interacts with the DefaultCreditorAutomation.sol.
These 2 functions calls the _validateLoan internal function at L70 and L109 (DebtorNFT.sol)

Inside the _validateLoan at L202 it checks that the balance of the vault should be exactly the debtAmount which was set during the initialization of the loan.
An attacker can send the minimum amount of debt token directly to the vault contract and make this line revert, and because of this the _validateLoan would always revert since the balance of the vault would never equal debtAmount now.

Recommendation:

Instead of relying on balanceOf() have an internal accounting system or change the condition from a strict equality to less than comparison.

HIGH-2 | RESOLVED

Incorrect Fee Accrual In Case Of A Referral

The activateLoan() function inside LoanVault.sol is called by the DebtorNFT contract when a loan is activated . Inside the function the 'fee' calculated at L114(in LoanVault.sol) , let's say this fee was calculated to be 100.

Just after fee calculation there is a check to see if there was a referral fee , if there was then referral fee is calculated as a part of the fee , let's say this comes out to be 20and this is paid to the referral address. Then fee is readjusted at L118 to exclude the referral fee , so in our example it would be 100 - 20 = 80 , this is the fee now that should be paid out.

Finally when the _takeFee is called at L120 instead of using the new updated 80 value as fee , fee is calculated again (which will be 100 again) and is paid out.
This means instead of paying 100 as fees , we paid 120 as fees.

Recommendation:

Change the _takeFee(collateral, (collateralAmount_ * _fees.initialisationFee) / 10000); to _takeFee(collateral, fee);
deductions, is transferred.

MEDIUM-1 | ACKNOWLEDGED

No Price Staleness Check

The function latestRoundData() has been used to fetch the price of an asset, but there are no price staleness checks.
Instances:
L24 → AggregatedChainnlinkOracle.sol
L25 → WBTCOracle.sol

There should be checks for the roundId and timestamp , i.e.

```
(uint80 roundID, int256 answer, , uint256 timestamp, uint80 answeredInRound) =
_btcUsdFeed.latestRoundData();
require(answeredInRound >= roundID, "Stale price");
require(timestamp != 0,"Round not complete");
```

Recommendation:

Introduce price staleness checks.
Comment: The client has suggested they can monitor their oracles for the StalePrice event and replace the oracles asap without disrupting the loans

MEDIUM-2 | RESOLVED

Unlimited Token Minting for CreditUSD and CSProtocolToken

The registeredMinters can mint an unlimited amount of tokens via mint(), if they are also authorized to call the onlyAuthorisedRole functions such as updateMintLimit(). Loss of private keys or collusion between the registeredMinters and onlyAuthorisedRoles can lead to unlimited token minting as well. This can lead to the CreditUSD token losing its peg.

Similarly, CSProtocolToken contract's mint() function can be used by a malicious admin to mint unlimited tokens.

Recommendation:

This can be detrimental to the project and can cause the token to lose its value. It is advised to use multisig wallet with at least 2/3 or 3/5 configuration for the contracts above. Additionally, consider adding a supply cap for CSProtocolToken to avoid unlimited minting of tokens.

Comments: The client said that the mint functions will only be callable by contracts such as the CreditUSDMinter or the DebtStaking contracts. No EOA, multisig or governance would be granted the rights to mint. However the rights to add authorised minters will be granted to a multisig or governance module with a sufficient number of signers.

MEDIUM-3 | RESOLVED

Reentrance in mint() of CreditUSDMinter

The mint() function of CreditUSDMinter violates checks effects interactions pattern. The external call of safeTransferFrom is made on the line: 68, which is before all the state changes in the state variables. This is not advised, because external calls before state changes can lead to reentrancy attacks. The safeTransferFrom contains call to the _checkOnERC721Received() which passes the call to untrusted contracts. This can lead to reentrancy exploits. However, this would just be a griefing attack in this case, as the attacker carrying out this exploit would lose more than gaining anything.

Recommendation:

It is still advised to use checks effects interactions pattern or use Reentrancy Guard from Openzeppelin to avoid cross function or cross-contract reentrancy exploits.

MEDIUM-4 | RESOLVED

Reentrance in CreditorNFT

The Checks Effects interactions (CEI) is not being followed in _deposit() of the CreditorNFT contract. This is because the safeTransferFrom is called on line: 182 on the debt token. If the debt token is an ERC20 token that implements hooks like beforeTransferFrom() or afterTransferFrom() which transfer calls to untrusted contracts, then this could result in reentrancy. Again this is would be a griefing attack for the attacker as he would be losing more than gaining.

Recommendation:

It is still advised to use checks effects interactions pattern or use Reentrancy Guard from Openzeppelin to avoid cross function or cross-contract reentrancy exploits.

LOW-1 | ACKNOWLEDGED

Liquidation In LounVault.sol should have a onlyCreditor Modifier

The liquidate() function inside LoanVault.sol is called via liquidate() function inside CreditorNFT.sol . Inside liquidate of CreditorNFT the debt tokens are sent to the vault , then liquidate of LoanVault is executed.
It is possible to call the liquidate function in LoanVault directly since there is no onlyCreditor modifier check , though it will not cause any harm it will require the liquidator to send the debt tokens via a direct transfer . It would be better to have a onlyCreditor modifier on the LoanVault's liquidate()

Recommendation:

Make the liquidate() function inside LoanVault.sol only callable by the CreditorNFT.sol

LOW-2 | ACKNOWLEDGED

Liquidations Might Not Be Profitable

The liquidation inside LoanVault.sol might not be profitable for the liquidator , the liquidator needs to provide the complete debt of the loan and will only get a portion of the collateral asset(minus the protocol and creditor fee) in return as award .

Recommendation:
Have a auto liquidation process which is done by a privileged role or increase the incentives for the liquidator.

LOW-3 | ACKNOWLEDGED

Ensure No Rounding

The calculation for rewardPerToken at L53 in RewardDistributor.sol might be subject to rounding if supply > reward*PRECISION.

Recommendation:
Ensure the above case is not possible.

LOW-4 | RESOLVED

Missing zero address checks for CreditUSDMinter

There is missing zero address check for creditUSD_ and controller_ parameter of function initialize(). Also there is missing zero address check in the setCollateralCreditLimit() function for collateralAsset parameter. This can lead to CreditLimit of zero address being set, which is a logical error. It can also lead to unintended or undiscovered issues. Similarly, there is missing zero address check for backingAsset parameter of the setBackingDeduction() function.

Recommendation:
It is advised to add a zero address check for the above functions.

Comments: The client added zero address check for creditUSD_ but not controller as they said it would not be required as it would revert if set to 0.

LOW-5 | ACKNOWLEDGED

Missing zero address and sanity checks in LoanVault

There is missing zero address check for controller parameter in initialize() function. Once set incorrectly, it cannot be set again. Also missing sanity checks for interestRate and liquidationThreshold_. If incorrectly assigned, it cannot be set again. Additionally, there is missing zero value check for minDuration_. Again If it is incorrectly set, it cannot be set again.

Recommendation:
It is advised to add above require checks for the same.

LOW-6 | RESOLVED

Missing zero address checks in ProtocolController

There is missing zero address check for initialAdmin and feeCollector_ parameter in initialize() function. Once set, it cannot be set again.

Recommendation:
It is advised to add a zero address check for the above functions.

Comments: The client said that the feeCollector can be set to the zero address explicitly, in which case no fees are collected (that's an easy way to turn off all fees). This implementation was missed in some cases, fixed in commits 6e4b7cf and 4eec52a

LOW-7 | ACKNOWLEDGED

Missing zero address check in CreditUSD

In CreditUSD contract, there is missing zero address check for minter parameter in updateMinter() and collateralAddress parameter in updateMintLimit() functions.

Recommendation:
It is advised to add missing zero address require check for the same.

LOW-8 | RESOLVED

Missing sanity checks in RewardAuctionHouse

There is Missing sanity checks for parameters in initialize() function. Here the minimumBid can be accidentally set to zero, or the auctionDuration can be zero as well as the beneficiary be set to zero address. This could lead to issues as these parameters can not be reset.

```
        minimumBid = minimumBid_;
        auctionDuration = auctionDuration_;
        beneficiary = beneficiary_;
```

Recommendation:

It is advised to add sufficient sanity checks for the above parameters.

INFORMATIONAL-1 | RESOLVED

Liquidations Might Revert If Collateral Is A Token Which Reverts On 0 Value Transfer

The collateral token can be any token whitelisted . When liquidation is called in the LoanVault.sol at L159 , withdrawableInterest is calculated at L160 and if withdrawableInterest is more than total collateral in the vault then withdrawableInterest = collateralBalance at L170 , let's say this was true and now withdrawableInterest = collateralBalance.
Because of this remainingCollateral calculated at L172 will be 0 and so will be the protocolFee and creditorFee , then we do _takeFee at L181 which in this case will do a 0 value transfer to the fee receiver and if collateral token reverts on 0 value transfer then the liquidation flow reverts .
Therefore in such a case bad position won't get liquidated.

Recommendation:

Have a check that if the value is more than 0 only then _takeFee.

INFORMATIONAL-2 | ACKNOWLEDGED

USDC Blacklisting Impact on LoanVault Liquidation Process

Description:

The `LoanVault` contract in the system is designed to handle collateral and debt for loans, potentially including the use of USDC as either collateral or debt. USDC, being a regulated stablecoin, includes a blacklisting feature allowing the USDC contract to prevent certain addresses from executing transactions. This feature can significantly impact the liquidation process of loans within the `LoanVault` contract if USDC is used.

Scenario:

1. **Liquidation Failure:** When the liquidation process is initiated, the contract attempts to transfer USDC (either as part of returning collateral to the borrower or moving debt to the liquidator). However, due to the blacklisting, the USDC transfer reverts.
2. **Resulting Impact:** The entire liquidation transaction fails, leaving the loan in a state where it can neither be repaid nor liquidated. This scenario leads to a deadlock, potentially causing financial loss and operational issues within the platform.

Recommendation:

Several approaches can be considered:

- **Pre-Liquidation Checks:** Implement checks within the LoanVault contract to verify whether the involved addresses are blacklisted in the USDC contract before initiating liquidation.
- **Fallback Mechanisms:** Develop a mechanism where alternative actions are taken if a USDC transfer fails due to blacklisting. This could involve using another form of collateral or a secondary process for dealing with blacklisted addresses.

INFORMATIONAL-3 | ACKNOWLEDGED

Asset Allowlist Check in `LoanVault`:

Description:
The `LoanVault` checks the debt asset against an allowlist, but this is not checked in the `CreditorNFT`.

Scenario: An unapproved asset could be passed to `LoanVault`, leading to a failed transaction after several steps have been executed.

Recommendation:
Perform the allowlist check earlier in the `_deposit` function to avoid unnecessary operations with unapproved assets.

INFORMATIONAL-4 | ACKNOWLEDGED

Careful Usage of Upgradeable contracts

Upgradeable contracts have been used throughout the codebase. For example, In CreditUSD contract, the Upgradeable ERC20 has been used. And in CreditUSDMinter, ERC721HolderUpgradeable has been used. Generally it is not recommended to use upgradeable contracts as it is against the notion of immutability of code. This can introduce trust issues with users as a malicious or compromised admin can change the code if it is an upgradeable contract. Or there can be unaudited changes to the code that could unintentionally introduce new bugs in the code.

Recommendation:
It is advised to carefully use the upgradeable contracts and use multisig wallets for the admin. It is also advised to audit the code changes done before the upgradation of the contracts.

INFORMATIONAL-5 | **UNRESOLVED**

Make Sure That Timelock Contract Is Deployed As Part Of The Deploy Script

If the timelock contract has not been deployed yet providing the appropriate roles , then the proposals won't get queued/executed unless the role is granted to the contract.

Recommendation:
Make sure the CSTimelock.sol is deployed as part of the deployment script.

INFORMATIONAL-6 | **ACKNOWLEDGED**

Allowing Fee-On-Transfer Tokens Might Be Problematic

It is possible that the creditor whitelists a fee-on-transfer token as a allowed collateral token.
These tokens might be problematic in accounting and result in wrong event emissions. For example , inside DebtorNFT.sol at L151 the debtor transfers "amount" of collateral to the vault , but in case of a fee-on-transfer token the amount that will be sent to the vault will be less than the amount , it might be possible that due to this the position becomes subject to liquidation.

When adding collateral to a vault the addCollateral function is called inside DebtorNFT.sol at L123 , the token transfer is done at L126 and in case of a fee on transfer token the event emission will be incorrect since it indexes 'amount' whereas the actual amount deposited would be lesser.

Recommendation:
Either don't allow such tokens or have proper accounting for such tokens.

INFORMATIONAL-7 | ACKNOWLEDGED

Resolve TODOs

Throughout the code base there are instances of TODOs and even empty contracts , these should be resolved before deployment. This includes →

TODO at L31-L51 in NFTBase.sol

RenderNFT.sol

L216 in LoanVault.sol

Recommendation:
Resolve the above

INFORMATIONAL-8 | RESOLVED

CEI Violation

There are instances in the codebase where the checks effects interaction pattern has not been followed . These include →

a.) L58 of RewardAuctionHouse.sol , there are state changes that occur after this transfer at L68 and L69

b.) L117 of LoanVault.sol , state is changed at L118 after the transfer.

	./creditUsd/CreditUSD.sol ./creditUsd/CreditUSDMinter.sol ./util/DefaultCreditorAutomation.sol ./nft/DebtorNFT.sol ./nft/DebtorStaking.sol ./nft/CreditorNFT.sol ./nft/StakingPool.sol ./nft/NFTBase.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./oracle/BaseOracleUSD.sol ./oracle/AggregatedChainlinkOracle.sol ./oracle/WBTCOracle.sol ./LoanVault.sol ./ProtocolController.sol ./governance/token/VeCreditSwapToken.sol ./governance/token/CSProtocolToken.sol ./governance/token/lib/TokenWrapper.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./governance/token/lib/TokenCheckpointer.sol ./governance/RewardAuctionHouse.sol ./governance/RewardDistributor.sol ./governance/CSGovernor.sol ./governance/CSTimelock.sol ./lib/TimestampLib.sol ./lib/Beacon.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./lib/BeaconProxy.sol **./lib/AccessControl.sol** **./lib/RenderNFT.sol** **./lib/Staking.sol** **./CreditSwapMarket.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Creditswap in verifying the correctness of their contracts code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the code's functionality and a review of the Creditswap contracts requirements for details about issuance amounts and how the system handles these.

```
    Setting up Auction House State
    ✓ Initialize Should Not Be Accesible Again (1621ms)
    ✓ Should Allow Access to Change Update Auction Duration
    ✓ Should Not Allow Access to Change Update Auction Duration
    ✓ Ensure Auction Reverts On Duplicate (344ms)
    ✓ Should Start Auction (49ms)
    ✓ Should Start Auction and Check for currentAuction (62ms)
    ✓ Should Start Auction and Bid (162ms)
    ✓ Should Be Able to Close Auction  (116ms)
    ✓ Should Be Able to Close Auction Through Start (116ms)
    ✓ Should Close and Withdraw From Auction  (207ms)
    Setting up Auction House State
    ✓ Should Not Be Able To Reinitalize (524ms)
    ✓ Should Revert With Zero Supply and Reward (171ms)
    ✓ Claim Should Revert If No WETH in Contract
    On
    ✓ Should Allow a New updatePeriodDuration (190ms)
    ✓ Should Revert If already Claimed (285ms)
    ✓ Should Have Proper Checks of claimableRewardOf (294ms)
Setting up Governor State
    Setup
    ✓ Should Check The Initialized Values (1300ms)
    Business Logic Voting
    ✓ Checking View Functions Return Values
    ✓ Ensure Proposal Reverts On Incorrect Description (111ms)
    ✓ Ensure GovernorNonexistentProposal Triggers (108ms)
    ✓ Ensure Proposal And Casting Vote Triggers Properly (163ms)
    ✓ Should Be Able To Vote on Proposal (289ms)
    ✓ Should Be Able To Vote on Proposal (342ms)
    ✓ Should Cancel (289ms)
```

```
        ✓ Should Queue and Execute (776ms)
        ✓ Trigger Executor Role
      Setting up Protocol Controller State
        ✓ Should Not Reinitialize
        ✓ Should AssetAllowList to be Updated by Authorized Role (79ms)
        ✓ Should Not Allow Unauthorized Users to make a call to fn with onlyAuthorisedRole Modifier  (165ms)
        ✓ Should Allow For New Referall Address (77ms)
        ✓ Should Allow For New Protocol Fees
        ✓ Should Allow For New setCreditorNFTAddress (69ms)
        ✓ Should Allow For New setDebtorNFTAddress (53ms)
        ✓ Should Allow For New oracleFor (85ms)
        ✓ Should Allow For New Role for Function (46ms)
        ✓ Should Allow For Checking of Roles (82ms)
    Setting Up Protoccol Token State
      Initializing Protocol Token
        ✓ Protocol Token Deployed Without Being Zero Address (2691ms)
        ✓ Should Revert If trying a non-authorized minter
        ✓ Should Update User to be Authorized (41ms)
        ✓ Minting Tokens to test Address (62ms)
        ✓ Revert if unauthorized user attempts to mint
        ✓ Should not be able to reinitialize
        ✓ Should not be able to reinitialize from testutils
    Setting Up VE Token State
      Testing Setup Values
        ✓ Function CLOCK_MODE should return mode=timestamp
        ✓ Function clock should return blockTime
        ✓ Function clock should revert if not equal to blockTime
        ✓ Testing Mock Mint Function
      Testing Voting Logic
        ✓ Get Past Votes Should be correct for msg.sender
        ✓ Ensure Tokens Have Been Deposited And Total Supply is Not Zero (123ms)
        ✓ Get The Votes of Account (76ms)
        ✓ Should get the proper amount of votes from the last block (256ms)
      Locking Business Logic
        ✓ Revert When Creating A second Lock (89ms)
        ✓ Slope Should Not Return Zero (179ms)
        ✓ Ensure getPastTotalSupply Tracks Supply (193ms)
        ✓ Ensure getPastTotalSupply Returns Zero on Epoch 0
        ✓ Ensure _votesAt Returns Zero on Epoch 0
        ✓ Ensuring Binary Search Tree Works (4107ms)
        ✓ Should Revert With Malicous Unlock Time Request (77ms)
        ✓ Should Properly Increase Time Locked (249ms)
        ✓ Transfer Withdrawn Funds Back to Msg.Sender (296ms)
        ✓ Should Deposit For Another User (260ms)
    ✓ Check Nonces and Delegates (72ms)
Setting up Credit Market State
    Setup Values
    ✓ Should not be able to reinitialize
    ✓ Ensure Values are Correctly Input
    Testing Authorized Roles
    ✓ Should Allow Authorized User to setAllowedToken
    ✓ Should Allow Authorized User to remove setAllowedToken (58ms)
    ✓ Should Revert If Not an NFT Contract  (60ms)
    ✓ Should Update Allowed NFT Contract  (42ms)
    ✓ Should Allow Fees To Update If Authorized User  (45ms)
    Business Logic Functionality
    ✓ Should Create Offer and accept Ether (186ms)
    ✓ Should Create Offer and accept Token (194ms)
    ✓ Should Revert With Unauthorised Signature (96ms)
    ✓ Should Revert With DutchAuctionDisallowed  (111ms)
    ✓ Should Have Offer.seller be msg.sender  (207ms)
    ✓ Should Have Offer.buyer be msg.sender  (141ms)
    ✓ Should Have Higher Start Price Without Reverting (148ms)
    ✓ Should Cancel (63ms)

    77 passing (19s)


CreditorNFT
    initialize
    ✓ should revert Already initialized
    mint
    ✓ mint
    ✓ should revert with InvalidDebtAmount
    invalidateOffer
    ✓ should revert with NonceAlreadyUsed
    ✓ invalidateOffer
    withdrawOffer
    ✓ withdrawOffer (2227ms)
    ✓ withdrawOffer (2315ms)
    ✓ should revert with VaultDoesNotExist
    ✓ should revert with Unauthorised
    ✓ claimInterest (2484ms)
    ✓ should revert with Unauthorised
    ✓ withdrawFunds (1928ms)
    setAllowedCollateral
    ✓ setAllowedCollateral
```

mintWithDebtorSignature

✓ mintWithDebtorSignature
✓ mintWithCreditorSignature
✓ should revert NonceAlreadyUsed
✓ should revert with expired

liquidate

✓ it should liquidate successfully (88ms)
✓ it should withdrawfunds successfully

AggregatedChainlinkOracle

latestPrice

✓ latestPrice (56ms)

CreditUSD

updateMinter

✓ updateMinter

updateMinter

✓ updateMinter

updateMintLimit

✓ updateMintLimit

mint

✓ mint (66ms)
✓ mint
✓ should fail mint
✓ burn
✓ burn should revert

CreditUSDMinter

setMintStatus

✓ setMintStatus

setMintStatus

✓ setMintStatus

setCollateralCreditLimit

✓ setCollateralCreditLimit

setBackingDeduction

✓ should set backing deduction

outstandingDebt

✓ should get outstanding debt
✓ should burn
✓ should revert

mint

✓ should mmint (3022ms)
✓ shoulld revert MintingDisabled

✓ should revert LoanInactive
✓ should revert with InvalidAsset
✓ should revert

Debtor Staking

setYearlyReward
✓ setYearlyReward (41ms)
replaceTokenRewardPool
✓ replaceTokenRewardPool (87ms)
claimRewards
✓ claimRewards (72ms)
✓ claimRewards
unstake
✓ should stake (1123ms)
✓ should unstake
✓ unstake should revert
✓ should force unstake (1727ms)
✓ should force unstake (1713ms)
✓ forceUnstaking should revert Unauthorised
✓ should revert if pool is address zero
✓ debtorstakinginit (1659ms)

loanvault

initialize
✓ should revert InvalidLiquidationFee
✓ initialize
✓ should revert InvalidLiquidationFee
✓ should revert InvalidLoanDuration
✓ should revert InvalidAsset
withdrawOffer
✓ should withdraw offer (48ms)
✓ should claim interest (1584ms)
✓ should repay (1914ms)
✓ should withdraw (1434ms)
collateralValue
✓ should return correct collateral value
✓ should return correct fees
✓ should get claimed interest (1728ms)
activateLoan
✓ should update loan status to activate (1579ms)
✓ should revert with InsufficientCollateral
✓ should revert with InvalidAsset
✓ activateLoan InvalidAsset

```
    ✓ should liquidate liquidate (1868ms)
    ✓ liquidate should fail
    ✓ liquidate InvalidDebt
    ✓ should not liquidate
    ✓ should not liquidate
    ✓ should liquidate (1944ms)
    ✓ liquidate should revert InvalidDebt
    ✓ should return correct minDuration

StakingPool
    updateYearlyReward
    ✓ updateYearlyReward (106ms)
    ✓ updateYearlyReward (64ms)

StakingPool
    initialize
    ✓ initialize
    updateYearlyReward
    ✓ updateYearlyReward (70ms)
    ✓ updateYearlyReward (71ms)
    fundRewards
    ✓ fundRewards (138ms)
    stake
    ✓ stake (121ms)
    ✓ stake
    ✓ stake

TImestamp Lib
    ✓ should set timestamps (43ms)
    ✓ should get end timestamp
    ✓ should get start timestamp

WBTCOracle
    latestPrice
    ✓ gets latest price (2761ms)
    DebtorNFT
    ✓ should mint token
    ✓ addCollateral should revert with NonceAlreadyUsed
    ✓ should invalidate offer
    ✓ invalidateOffer should revert NonceAlreadyUsed
    ✓ should repay
    ✓ repay should revert with Unauthorised
    ✓ should add collateral
```

```
mintWithDebtorSignature
✓ mintWithDebtorSignature should revert with ConfigMismatch
✓ mintWithDebtorSignature should revert with ConfigMismatch
✓ mintWithDebtorSignature should ConfigMismatch
✓ should mint With Debtor Signature
✓ should mint With Debtor Signature
✓ mintWithDebtorSignature should revert expired
✓ should mint mint From Automation
✓ should revert with ConfigMismatch

104 passing (3m)
```

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	%UNCOVERED LINES
AggregatedChainlink Oracle.sol	100	100	100	100	
BaseOracleUSD.sol	100	100	100	100	
WBTCOracle.sol	100	100	100	100	
CreditUSD.sol	100	100	100	93.75	
CreditUSDMinter.sol	97.37	92.86	100	98.08	
TimestampLib.sol	100	100	100	100	
DebtorStaking.sol	100	77	100	100	
Staking.sol	100	66.67	90	94.59	
StakingPool.sol	96.3	85	75	93.55	
Beacon.sol	100	100	100	100	
NFTBase.sol	100	75	100	100	
BeaconProxy.sol	100	100	100	100	
VeCreditSwapToken.sol	100	92.86	100	97.3	

CSProtocolToken.sol	100	100	100	100
TokenCheckpointer.sol	98.18	87.5	92.31	98.91
CSTimelock.sol	100	100	100	100
CSGovernor.sol	100	100	100	100
RewardDistributor.sol	96.43	77.78	100	100
RewardAuctionHouse.sol	97.37	64.29	100	100
TokenWrapper.sol	97.62	64.71	80	98.04
ProtocolController	100	82.35	100	100
CreditSwapMarket.sol	98.51	64.81	100	100
CreditorNFT.sol	88.1	95.45	100	87.76
DefaultCreditorAutomation.sol	93	82.31	100	95.26
DebtorNFT.sol	89	95.45	100	89
LoanVault.sol	90.48	63.64	90	90
AccessControl.sol	80	60	83.33	87.5
RenderNFT	100	100	100	100
All Files	**97.22**	**86.70**	**96.80**	**97.27**

We are grateful for the opportunity to work with the Creditswap team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Creditswap team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.