ENIGMA DARK

Securing the Shadows

Security Review

Credit Coop v2

October 30, 2024

Contents

Summary

Enigma Dark
Enigma Dark is a web3 security firm leveraging the best talent in the space to secure all kinds of blockchain protocols and decentralized apps. Our team comprises experts who have honed their skills at some of the best auditing companies in the industry. With a proven track record as highly skilled white-hats, they bring a wealth of experience and a deep understanding of the technology and the ecosystem.

Learn more about us at enigmadark.com

Credit Coop v2
Credit Coop v2 offers smart contracts that escrow future cash flows, enhancing efficiency. Credit Coop transforms future cash flows into instant liquidity. Borrowers enjoy non-dilutive financing, flexible terms, and automated repayments. Lenders gain access to diverse and secured receivables with real-time transparency.

Engagement Overview

Over the course of 4 weeks, the Enigma Dark team conducted a security review of the Credit Coop v2 project. The review was performed by 0xWeiss & Ladboy.

The following repositories were reviewed at the specified commits:

Repository	Commit
credit-cooperative/Line-Of-Credit-v2	`15059e06de9ed141ea0bfdbe13b4d98ffa7cb7a0`
credit-cooperative/Vaults	`314de654b0f1b2d6428a99725c6a95360dd24060`

Risk Classification

Severity	Description
High	Exploitable, causing loss or manipulation of assets or data.
Medium	Risk of future exploits that may or may not impact the smart contract execution.
Low	Minor code errors that may or may not impact the smart contract execution.

Vulnerability Summary

Severity	Count	Fixed	Acknowledged
High	11	11	0
Medium	13	12	1
Low	28	19	9
Informational	5	5	0

Findings

Index	Issue Title	Status
H-01	The `close` function in SpigotedLine does not charge service fee / over-charge the interest payment / revert.	Fixed
H-02	Consider using TWAP to get sqrtRatioX96 in Uniswap V3 Oracle	Fixed
H-03	The `sync` function in CreditStragety can revert because of out of bounds index access	Fixed
H-04	Malicious univ3Manager can cause loss of funds	Fixed
H-05	A malicious beneficiary can DOS or steal funds from the Spigot contract.	Fixed
H-06	The `pullTokens` function in Spigot has no access control	Fixed
H-07	Deposit amount should be tracked properly for ERC4626 assets	Fixed
H-08	The liquidation action in the `Escrow` contract does not check if the line of credit is subject to liquidation.	Fixed
H-09	The default calldata for the `reduceLiquidity` function should not execute when the borrower removes liquidity in `LaaSEscrow`	Fixed
H-10	Missing slippage protection when interacting with Angles transmuter	Fixed
H-11	The swap function in `StablecoinStrategy` is missing slippage control	Fixed
M-01	Flawed and risky approval system	Fixed
M-02	`proratedOriginationFee` is overpaid on refinancing	Fixed
M-03	Borrower address is not on sync between Line and Escrow	Fixed

M-04	Consider adding reentrancy protection in `sweep` from the `SpigotLine` contract	Fixed
M-05	Management fee is over estimated when the vaults are paused	Fixed
M-06	Spitgot lender repayment can be DOsed via 1 wei token transfer.	Fixed
M-07	Unused approval is not reset after external call in LaasEscrow	Fixed
M-08	Increase liquidity and remove liquidity in LaasEscrow does not validate lp function signature	Fixed
M-09	Incorrect WETH address used	Fixed
M-10	Usage of a single DEX for token swaps is suboptimal	Acknowledged
M-11	Malicious borrower can use a blacklisted address to block lender from withdrawing funds early	Fixed
M-12	The `RefinanceCredit` function from LineOfCredit does not accure pending interest before changing the rate rate	Fixed
M-13	`reallocateFunds` logic in the StablecoinStrategy is broken	Fixed
L-01	Consider to de-couple `claimOwnerTokens`, `swap`, and `repayment` in `SpigotedLine`	Fixed
L-02	`mint()` should have access control	Fixed
L-03	`accrueInterest()` can be DOSed if there are too many credit lines opened at the same time	Acknowledged
L-04	`proposalCount` can be artificially inflated	Fixed
L-05	Nonce and deadline missing on "abort" and "setFees"	Acknowledged
L-06	Incorrect permissions for registering contracts and whitelisting functions	Fixed
L-07	if the default beneficiary is updated to an EOA account, the borrower can't call `resetBeneficiaries` and	Acknowledged

	`updateBorrowerContractSplit`	
L-08	Proposals are kept open even when borrower gets updated	Fixed
L-09	Overall griefing capacity between lenders and borrowers	Acknowledged
L-10	Vault managers can be griefed on executions	Acknowledged
L-11	Do not hardcode slippage while swapping	Fixed
L-12	`WithdrawAll()` can fail to 0 asset withdrawals	Fixed
L-13	Consider whitelisting the token0 + token1 + fee tier for uniswap v3 nft in the `Escrow` contract.	Fixed
L-14	If the `accrueManagementFee` function is not called for a prolonged period of time, accrueManagementFee can revert.	Fixed
L-15	Code comment and implementation is out of sync when swapping credit positions	Fixed
L-16	Borrower is not capable of repaying lender in Spigot.sol	Fixed
L-17	Whilelisted smart contract and method in Escrow contract cannot be removed.	Fixed
L-18	Chainlink oracle does not check if the sequencer is down in L2 network.	Acknowledged
L-19	Oracle returned value is not sufficiently validated.	Acknowledged
L-20	Claimable fund should be used to repay the debt first when the line of credit contract is subject to liquidation.	Fixed
L-21	OTC swap should exclude the swap fee amount when computing the price impact	Acknowledged
L-22	OTC swap assumes that all stablecoin are strictly 1:1 pegged	Acknowledged
L-23	`earlyWithdrawalFee` when adding credit is not capped	Fixed

L-24	Balances are updated before token transfers (ANTI PATTERN)	Fixed
L-25	Incorrect symbol for credit position tokens	Fixed
L-26	The `depositAndRepay` function in LineOfCredit should accrue interest first	Fixed
L-27	LendingVault is not capable of calling the `depositIntoUSDA` function in USDAStrategy	Fixed
L-28	An old pool will still have max approval after the pool address is changed in WETHStrategy contract	Fixed
I-01	Unused code across the repository	Fixed
I-02	NATSPEC issues	Fixed
I-03	CreditStrategySet event should take the parameter `creditStrategy` not `manager`	Fixed
I-04	No need to approve receiver address when swap credit position	Fixed
I-05	SmartEscrow is not capable of handling native ETH	Fixed

Detailed Findings

High Risk

H-01 - The `close` function in SpigotedLine does not charge service fee / over-charge the interest payment / revert.

Severity: High Risk

Description:

`SpigotedLine` overrides the `close` function from the `LineOfCredit` contract, however, if we compare the overriden function with the LineOfCredit `close` function, we can see that there are some issues:

```
    function close(uint256 id) public virtual {
        _onlyBorrowerOrServicer();
        Credit storage credit = credits[id];
        _accrue(credit, id);
        uint256 totalOwed = credit.interestAccrued +
FeesLib._calculateServicingFee(fees, credit.interestAccrued);

        // msg.sender clears facility fees and close position
        _repay(credit, id, totalOwed, msg.sender);
        _close(credit, id);
    }
```

- Issue 1: The `repay` function inside `SpigotedLine` does not charge a service fee inside the following `if` statement:

```
          if (unusedTokensForClosingPosition >= facilityFee) {
                 // clear facility fees and close position
                 unusedTokens[credit.token] -= facilityFee;
                 _repay(credit, id, facilityFee, address(0));
                 _close(credit, id);
             }
```

- Issue 2: When the payer address is `address(this)` this code will execute and revert.

```
        // if we arent using funds from reserves to repay then pull tokens
from target
        if (payer != address(0)) {
            // receive tokens from payer
            LineLib.receiveTokenOrETH(credit.token, payer, amount);
        }
```

The following will revert:

```
ERC20(token).transferFrom(address(this), address(this), amount);
```

- Issue 3: Inside the `else` block, the code tries to `repay`, then LineOfCredit `close` will charge interest again.

```
               unusedTokens[credit.token] -= unusedTokensForClosingPosition;
                _repay(credit, id, unusedTokensForClosingPosition,
address(this));
                LineOfCredit.close(id);
```

Recommendation:

We think that if the code just tries to use `unusedTokens[credit.token]` to cover the interest payments, the following implementation should work. As seen below, the code uses `unusedTokens[credit.token]` to cover the `totalOwed` amount first.

```
    function close(uint256 id) public override {
        _onlyBorrowerOrServicerOrAdmin();
        Credit storage credit = credits[id];
        _accrue(credit, id);

       uint256 unusedTokensForClosingPosition = unusedTokens[credit.token];

        uint256 totalOwed =
            credit.interestAccrued + FeesLib._calculateServicingFee(fees,
credit.interestAccrued);

        if(unusedTokensForClosingPosition >= totalOwed) {
            unusedTokens[credit.token] -= totalOwed;
        } else {
            unusedTokens[credit.token] = 0;
            totalOwed -= unusedTokensForClosingPosition;
        }

        // msg.sender clears facility fees and close position
        _repay(credit, id, totalOwed, msg.sender);
        _close(credit, id);
    }
```

Developer Response: Fixed at commit `6924cae30319477dd8d5b9fa0b10fc0c4f930e28`.

H-02 - Consider using TWAP to get sqrtRatioX96 in Uniswap V3 Oracle

Severity: High Risk

Description:

This is the current code for uniswap v3 oracle.

The function parameters for `getAmountsForLiquidity` are:

```
    function getAmountsForLiquidity(
        uint160 sqrtRatioX96,
        uint160 sqrtRatioAX96,
        uint160 sqrtRatioBX96,
        uint128 liquidity
    ) internal pure returns (uint256 amount0, uint256 amount1) {
        if (sqrtRatioAX96 > sqrtRatioBX96) (sqrtRatioAX96, sqrtRatioBX96) =
(sqrtRatioBX96, sqrtRatioAX96);

        if (sqrtRatioX96 <= sqrtRatioAX96) {
            amount0 = getAmount0ForLiquidity(sqrtRatioAX96, sqrtRatioBX96,
liquidity);
        } else if (sqrtRatioX96 < sqrtRatioBX96) {
            amount0 = getAmount0ForLiquidity(sqrtRatioX96, sqrtRatioBX96,
liquidity);
            amount1 = getAmount1ForLiquidity(sqrtRatioAX96, sqrtRatioX96,
liquidity);
        } else {
            amount1 = getAmount1ForLiquidity(sqrtRatioAX96, sqrtRatioBX96,
liquidity);
        }
    }
```

All the parameters are passed in the correct order unless the `sqrtRatioX96` parameter:

```
  /// @param sqrtRatioX96 A sqrt price representing the current pool prices
token0Price, token1Price
```

The current pool prices should not be solely derived from `token0Price` and `token1Price`.

The updated implementation to get the `sqrt` price representing the current pool prices uses the `computeCurrentPoolPrice` function where `pool.slot0()` is used to query the spot price, and the code in else statement is used to query the TWAP price.

We can run the test after updating the oracle code:

Please note that we renamed the `test_can_add_position` in `UniV3Collateral.t.sol` to `test_can_add_position_poc`.

Command to run:

```
forge test -vv --match-test "test_can_add_position_poc"
```

Output:

```
Ran 1 test for contracts/tests/UniV3Collateral.t.sol:UniV3EscrowTest
[PASS] test_can_add_position_poc() (gas: 753044)
Logs:
  spotPriceX96:          41702742322088139591811 77
  twap PriceX96:         41702654841923869310523 01
  oldSqrtPriceX96:       4172136916887338445436588222192
  position ticker lower: 37343927387651759218673 50
  position ticker upper: 38712730979008435253465 35
  amount0: 0, amount1: 9954408631
  spotPriceX96:          41702742322088139591811 77
  twap PriceX96:         41702654841923869310523 01
  oldSqrtPriceX96:       4172136916887338445436588222192
  position ticker lower: 37343927387651759218673 50
  position ticker upper: 38712730979008435253465 35
  amount0: 0, amount1: 9954408631
  spotPriceX96:          41702742322088139591811 77
  twap PriceX96:         41702654841923869310523 01
  oldSqrtPriceX96:       4172136916887338445436588222192
  position ticker lower: 37343927387651759218673 50
  position ticker upper: 38712730979008435253465 35
  amount0: 0, amount1: 9954408631
  Collateral Value:  995858948262
```

As we can see, the old `SqrtPriceX96` is too large and not in the scale of `spotPriceX96` or twap `PriceX96`.

If the `oldSqrtPriceX96` is continue to be used, the else block will always be triggered and only the `amount1` token balance will be counted. This is because `sqrtRatioX96` is greater than both `sqrtRatioAX96` and `sqrtRatioBX96`,

Recommendation:

Use the following reference implementation:

https://github.com/revert-finance/lend/blob/37d6fa4b4806553ab3598e1bb47b1b24a5bb3abd/src/V3Oracle.sol#L362

The spot `pool.slot0()` can be manipulated, it is recommended to use the `TWAP`.

The `observe` function may fail when there is not enough history available.

The protocol should carefully evaluate the Uniswap pool being used to check if the liquidity is poor and use a pool with enough history.

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/pull/118

H-03 - The `sync` function in CreditStragety can revert because of out of bounds index access

Severity: High Risk

Description:

The `sync` function in CreditStragety modifies the token id array while looping over the token ids, which can result in an out of bounds error and a transaction revert.

```
    function sync() external {
        _onlyVaultManagerOrOperator();
        // iterate through all credit positions owned by the Vault to reset
totalAssets()
        address line;
        uint256 tokenId;
        address tokenContract;
        uint256 creditDeployedToPosition;
        address tokenOwner;

        uint256 index = 0;
        while (index < tokenIds.length) {
            tokenId = tokenIds[index];
            line = creditTokenIdToLine[tokenId];
            tokenContract = address(ILineOfCredit(line).tokenContract());
            creditDeployedToPosition = creditDeployedToPositions[line]
[tokenId];
            tokenOwner = this.ownerOfTokenId(tokenContract, tokenId);

            if (tokenOwner != address(this)) {
                totalCreditDeployed -= creditDeployedToPosition;
                creditDeployedToPositions[line][tokenId] = 0;
                emit ReduceCreditDeployed(line, tokenId,
creditDeployedToPosition);

                // remove tokenId from tokenIds array and
creditTokenIdToLine mapping
                _removeTokenId(index);
                index = 0;
                creditTokenIdToLine[tokenId] = address(0);
                continue;
            }

            index += 1;
        }
```

if `(tokenOwner != address(this))` case is hit, the index variable will be reset to 0, but we pop an element out of the tokenIds array.

In the case the token id array has three elements and loops run at index 0, 1, 2, this is the good case.

The bad case is:

```
loop runs at index 0, it is ok.

loop runs at 1, index goes to index 0

loop runs at index 0

loop runs at index 1

loop runs at index 2 => out of bounds access in tokenId array => transaction
revert.
```

Recommendation:

The key is to avoid modifying the token id array inside the while loop while we are iterating over the token id.

Developer Response:

Fixed at commit: https://github.com/credit-cooperative/Vaults/commit/d790daa4f24265c0fb8045576874f45ab9bf4783

H-04 - Malicious univ3Manager can cause loss of funds

Severity: High Risk

Description

In the `Escrow` contract, the uni manager can `mint` / `decrease` / `increase` / `burn` liquidity / `collect` fee.

But a malicious uni manager can cause a loss of funds in a few ways:

Exploit path one - increase liquidity for arbitrary token id.

Increase liquidity for a token id that does not belong to the contract.

In UniV3EscrowLib.sol#increaseLiquidity the code does not validate if the token id belongs to the contract.

The parameter: `INonFungiblePositionManager.IncreaseLiquidityParams` is not validated:

```
    struct IncreaseLiquidityParams {
        uint256 tokenId; // @audit <== token id is not validated.
        uint256 amount0Desired;
        uint256 amount1Desired;
        uint256 amount0Min;
        uint256 amount1Min;
        uint256 deadline;
    }
```

A malicious uni manager can mint a token id to another address and then use the funds in the escrow contract to increase liquidity for that token id.

Even when the collateral ratio is checked, the `increaseLiquidity` function will very likely reduce the collateral value.

Exploit path two - mint / decrease / increase / burn liquidity lack of slippage control.

For all these functions, the parameter `amount0Min` and `amount1Min` serve as slippage control.

For example, when decreasing liquidity, if the received token 0 is less than `amount0Mint` or the received token 1 is less than `amount1Min`, the transaction should revert.

But the code never validated the uni v3 manager supplied parameters.

A malicious uni v3 manager can set `amount0Min` and `amount1Min` to 0 so the transaction would suffer from slippage loss.

Recommendation

1- Validate if token id belongs to the contract when calling increase liquidity,

2- Validate slippage parameter when calling the uni v3 poisition functions.

or

Add the modifier `mutualConsent(admin, univ3Manager)` to all uniswap V3 functions to make sure the
`univ3Manager` cannot supply arbitrary data.

Developer Response:

fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/3ba5c3dcd0a814027822e24b7538ca3ef0e1c32c

The uniswap v3 manager can still choose to disable the slippage protection so a trustworthy uniswap manager should be selected by the borrower and the protocol.

H-05 - A malicious beneficiary can DOS or steal funds from the Spigot contract.

Severity: High Risk

Description

A malicious beneficiary can DOS or steal funds from the Spigot contract.

There are two functions that the beneficiary address can call:

`updateRepaymentFunc` and `updatePoolAddress`

Updating to a different repayment function or pool address will block the repayment when calling `_repayLender`

```
if (bytes4(functionData) != beneficiary.repaymentFunc) {
    revert BadFunction();
}
```

Updating to a different or malicious pool address can lead to theft of funds because the code gives an approval to the pool address when repaying the lender:

```
if (bytes4(functionData) != beneficiary.repaymentFunc) {
    revert BadFunction();
}

IERC20(beneficiary.creditToken).forceApprove(beneficiary.poolAddress,
amount);
```

Then, a malicious pool address can abuse this approval to transfer the funds out from the spigot contract.

Updating the pool address or modifying pool address logic to return a large amount of debt owned can screw the borrower tool.

```
    function _getDebtOwed(address lender) internal view returns (uint256) {
        if (_beneficiaryData(lender).debtVoided) return 0;
        // get the func
        bytes4 getDebtFunc = _beneficiaryData(lender).getDebtFunc;

        // call the func
        // NOTE: assumes that the only input is the borrower address and the
only output is the debt owed in uint
        address poolAddress = _distributionData().data[lender].poolAddress;
        (bool success, bytes memory data) =
poolAddress.staticcall(abi.encodeWithSelector(getDebtFunc, _borrower()));

        if (success) {
            uint256 debtOwed = abi.decode(data, (uint256));
            return debtOwed;
        }

        return 0;
    }
```

Because the code replies on [`_getDebtOwned`] to query the debt info from the beneficiary pool address.

Recommendation

add the modifier `mutualConsent(admin, beneficiary)` to this function `updateRepaymentFunc` and `updatePoolAddress`

Protocol Response Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/4ffd9fda6288c3fe5748371517e3b37961f5f7c5

H-06 - The `pullTokens` function in Spigot has no access control

Severity: High Risk

Description:

There is no access control in the `pullTokens` function, anyone can keep calling this function repeatedly to pull tokens the spigot contract.

Recommendation:

Add access control to the `pullTokens` function.

We think applying the `onlyOperatorServicerOrAdmin` modifier should be enough.

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/f49331e67abb5c76ac6229f706ab8dc2d4053aaa

H-07 - Deposit amount should be tracked properly for ERC4626 assets

Severity: High Risk

Technical Details:

The function _getCollateralvalue from the `Escrow`:

```
            uint256 unused;

            try ISpigotedLine(self.owner).unused(token) returns (uint256
_unused) {
                unused = _unused;
            } catch {
                unused = 0;
            }
            d = self.deposited[token];
            // new var so we don't override original deposit amount for 4626
tokens
            uint256 deposit = d.amount + unused; // <=======
            if (deposit != 0) {
                if (d.isERC4626) {
                    // this conversion could shift, hence it is best to get
it each time
                    (bool success, bytes memory assetAmount) =

token.call(abi.encodeWithSignature("previewRedeem(uint256)", d.amount));
                    if (!success) continue;
                    deposit = abi.decode(assetAmount, (uint256)) + unused;
// <=======
                }

                collateralValue +=
CreditLib.calculateValue(o.getLatestAnswer(d.asset), deposit,
d.assetDecimals);
            }
```

specifies that if the collateral asset is a ERC4626 vault, the `d.amount + unused` would be the share amount.

But, the code is missing to call `previewRedeem` to convert the share to amount.

Impact:

After the conversion, the deposit amount is updated if `d.isERC4626` is true.

```
   (bool success, bytes memory assetAmount) =
     token.call(abi.encodeWithSignature("previewRedeem(uint256)",
d.amount));
   if (!success) continue;
   deposit = abi.decode(assetAmount, (uint256)) + unused;
```

But the unused amount should be considered as share and not adding back to the deposit amount again.

Otherwise, we are using the asset amount + share amount and we over-value the collateral worth.

Recommendation:

Make the following updates:

For ERC20 tokens:

```
asset balance = balance of ERC20 token

collateral worth = one unit of asset price * asset balance for ERC20
```

For ERC4626:

```
asset balance = convert shares amount to asset balance (convertToAsset)

colalteral worth for ERC4626 = one unit of asset price * asset balance
```

Proposed fix:

https://github.com/credit-cooperative/Line-Of-Credit-v2/blob/2db2df91e82099edc653fd52be39d87dc8c9101a/contracts/modules/oracle/Oracle.s

```
 if (is4626) {
    address underlyingToken = abi.decode(tokenAddrBytes, (address));

    (bool successDecimalsVault, bytes memory decimalBytesVault) =
        token_.staticcall(abi.encodeWithSignature("decimals()"));

    if (!successDecimalsVault || decimalBytesVault.length == 0) {
        return NULL_PRICE;
    }

    (bool successDecimalsUnderlying, bytes memory decimalBytesUnderlying) =
        underlyingToken.staticcall(abi.encodeWithSignature("decimals()"));

    if (!successDecimalsUnderlying || decimalBytesUnderlying.length == 0) {
        return NULL_PRICE;
    }

    token_ = underlyingToken;
}
```

The code above takes an address of an ERC4626 token and extracts the underlying token address, and we set `token_ = underlying token`.

After, in the code below, the change can be kept simple:

https://github.com/credit-cooperative/Line-Of-Credit-v2/blob/15059e06de9ed141ea0bfdbe13b4d98ffa7cb7a0/contracts/utils/EscrowLib.sol#L77

```
    if (deposit != 0) {
        if (d.isERC4626) {
            // this conversion could shift, hence it is best to get it each
time
            (bool success, bytes memory assetAmount) =
                token.call(abi.encodeWithSignature("previewRedeem(uint256)",
d.amount));
            if (!success) continue;
            deposit = abi.decode(assetAmount, (uint256));
        }

        collateralValue +=
CreditLib.calculateValue(o.getLatestAnswer(d.asset), deposit,
d.assetDecimals);
    }
```

Developer Response: The proposed fix has not been followed, instead it has been fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/pull/119

H-08 - The liquidation action in the `Escrow` contract does not check if the line of credit is subject to liquidation.

Severity: High Risk

Technical Details:

The function below allows the admin to liquidate the uniswap v3 position nft.

```
    /**
     * @notice - if LIQUIDATABLE, the admin can send the uni v3 position to
any address
     * @dev    - callable by admin
     * @param tokenId - the id of the uni v3 position
     * @param to      - the address we send the uni v3 position to
     */
    function liquidatePosition(uint256 tokenId, address to) external
nonReentrant {
        _onlyAdmin();
        uniV3State.liquidate(tokenId, to);
    }
```

However, the code does not check if the line of credit position in an insolvent state and subject to liquidation.

Impact:

A health line of credit can be liquidated.

Recommendation:

Just like the liquidate ERC20 token function, add the status check

```
   function liquidate(uint256 amount, address token, address to) external {
        if (_getLineStatus() != LineLib.STATUS.LIQUIDATABLE) {
            revert NotLiquidatable();
        }
        _onlyAdmin();
        state.liquidate(amount, token, to);
    }
```

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/3eb9f5511ded798836a19ba288fe19e0d682ca66

H-09 - The default calldata for the `reduceLiquidity` function should not execute when the borrower removes liquidity in `LaaSEscrow`

Severity: High Risk

Technical Details:

The code below should reduce liquidity.

```
    function reduceLiquidity(
        EscrowState storage state,
        mapping(uint256 => ILaaSEscrow.LiquidityPosition) storage
liquidityPositions,
        uint256 index,
        address pool,
        uint256 minimumCollateralRatio,
        address borrower,
        bytes memory removeLiquidityData
    ) external {
        // only borrower or owner
        ILaaSEscrow.LiquidityPosition memory lp = liquidityPositions[index];
        if (lp.LpToken == address(0)) {
            // if LP position does not exist, revert
            revert ILaaSEscrow.NoLiquidityPosition();
        }
        uint256 cratio = Escrow(address(this)).getCollateralRatio();
        if (msg.sender != borrower && (minimumCollateralRatio + lp.apron <=
cratio)) {
            revert ILaaSEscrow.ApronNotBreached();
        }
        lp.apron = uint32(cratio - minimumCollateralRatio); // update apron
so we dont allow a reentrancy attack
        if (msg.sender == borrower) {
            _removeLiquidity(state, removeLiquidityData, index, lp, pool);
        }
        _removeLiquidity(state, lp.removeLiquidityData, index, lp, pool);
    }
```

As the comment outlines.

However, if the borrower calls the `reduceLiquidity` function, `_removeLiquidity` will run twice

Impact:

The function call to reduce liquidity with default calldata still runs when the borrower removes liquidity.

Recommendation:

Make the following change:

```
        if (msg.sender == borrower) {
            _removeLiquidity(state, removeLiquidityData, index, lp, pool);
        } else {
             _removeLiquidity(state, lp.removeLiquidityData, index, lp,
pool);
        }
```

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/54a420c5b5c9efa5712a07324b5657ed10838cdd

H-10 - Missing slippage protection when interacting with Angles transmuter

Severity: High Risk

Technical Details:

The `withdrawAll()` functions initially withdraws USDA from the stUSDA vault and then swaps the USDA to the strategy asset address via Angles transmuter contract:

```
    function withdrawAll() external {
        _onlyLendingVaultOrOwner();

        // Withdraw all assets from stUSDA vault
        uint256 maxWithdrawableUSDA =
IERC4626(stUSDA).maxWithdraw(address(this));
        uint256 sharesBurned =
IERC4626(stUSDA).withdraw(maxWithdrawableUSDA, address(this),
address(this));
        emit VaultWithdraw(address(stUSDA), address(USDA),
maxWithdrawableUSDA, sharesBurned);

        uint256 swapAmount = IERC20(USDA).balanceOf(address(this));
        uint256 amountOut =
            transmuter.swapExactInput(swapAmount, 0, USDA, address(asset),
address(this), block.timestamp + 60 minutes);
        emit StrategyExchange(USDA, address(asset), swapAmount, amountOut);
```

The issue is that the slippage protection parameter `amountOutMin` is set to 0 when swapping USDA for the asset of the strategy:

```
  uint256 amountOut =
            transmuter.swapExactInput(swapAmount, 0, USDA, address(asset),
address(this), block.timestamp + 60 minutes);
        emit StrategyExchange(USDA, address(asset), swapAmount, amountOut);
```

Impact:

When a withdrawal happens via `withdrawAll()`, the transaction may suffer loss from a sandwich attack.

Recommendation:

Do calculate an acceptable `amountOutMin` parameter to avoid getting sandwiched on withdrawals

Developer Response: fixed at commit: https://github.com/credit-cooperative/Vaults/commit/a06b8f465b477decaca55df31c1481f065325cc6

H-11 - The swap function in `StablecoinStrategy` is missing slippage control

Severity: High Risk

Technical Details:

The function `_exchange` in `StablecoinStrategy` aims to make a swap

```
    function _exchange(address _from, address _to, uint256 _amount) internal
override returns (uint256) {
        int128 fromIndex = crvPoolTokens[_from];
        int128 toIndex = crvPoolTokens[_to];

        uint256 balanceBefore = IERC20(_to).balanceOf(address(this));

        crv3Pool.exchange(fromIndex, toIndex, _amount, 1 wei);
        uint256 amountOut = IERC20(_to).balanceOf(address(this)) -
balanceBefore;
        emit StrategyExchange(_from, _to, _amount, amountOut);

        return amountOut;
    }
```

However, the slippage control is missing:

```
crv3Pool.exchange(fromIndex, toIndex, _amount, 1 wei);
```

The `min_dy` parameter is set to 1 wei, which means there is no slippage protection.

```
interface ICurvePool {
    function exchange(int128 i, int128 j, uint256 dx, uint256 min_dy)
external;
```

Impact:

When this `_exchange` function is triggered, the transaction may suffer loss from MEV sandwich attacks.

Recommendation:

It is recommended to add slippage control just like the code in the other contract.

```
    function swapThroughCRVPool(uint256 amount, bool wethToSteth) external {
        _onlyLendingVaultOrOwner();
        if (wethToSteth) {
            // The crv pool is ETH/STETH not WETH so we need to unwrap it
first
            WETH.withdraw(amount);

            // 0.5% slippage
            uint256 stETHOut = crvPool.exchange{value: amount}(0, 1, amount,
amount.mulDiv(MAX_SLIPPAGE, DENOM));
            emit StrategyExchange(address(WETH), address(steth), amount,
stETHOut);

            // In the event anything is unused, wrap it back to WETH
            uint256 remainingBalance = address(this).balance;
            if (remainingBalance != 0) {
                WETH.deposit{value: remainingBalance}();
            }
        } else {
            // 0.5% slippage
            uint256 wethOut = crvPool.exchange(1, 0, amount,
amount.mulDiv(MAX_SLIPPAGE, DENOM));
            emit StrategyExchange(address(steth), address(WETH), amount,
wethOut);
        }
    }
```

The `amount.mulDiv(MAX_SLIPPAGE, DENOM)` should compute the `min_dy` amount.

Developer Response: fixed at commit: https://github.com/credit-cooperative/Vaults/commit/4b7eb7c10745de82560b1d7557cce9ea269c33d4

Medium Risk

M-01 - Flawed and risky approval system

Severity: Medium Risk

Technical Details:

Most of the architecture relies on approving max (uint256) to smart contracts from inside credits core system, and from outside, such as Angle or Curve pools.

This is very risky and flawed in some instances as it can be seen in previous issues that approvals are sometimes not reset when for instance curve pools are updated, allowing for an unnecessary systemic risk of draining approvals.

Other flaws can be found for example in the `increaseLiquidity()` function in the `UniV3EscrowLib` library which approves max all the time which would be redundant in case a token is repeated:

```
IERC20(position.token0).forceApprove(address(self.nftPositionManager),
type(uint256).max);

IERC20(position.token1).forceApprove(address(self.nftPositionManager),
type(uint256).max);

INonFungiblePositionManager(self.nftPositionManager).increaseLiquidity(params
);
```

Impact:

Flawed and risky approval system

Recommendation:

All the spots were contracts are approving max to other contracts, should change to use `forceApprove()` for the exact amount, and after the function calls in the same transaction, `forceApprove()` again to 0.

This should be done at the very list for contracts that are not from the Line of Credit system.

Developer Response: fixed at commit: https://github.com/credit-cooperative/Vaults/pull/72

M-02 - `proratedOriginationFee` is overpaid on refinancing

Severity: Medium Risk

Technical Details:

An origination fee is charged when the first credit is created:

```
  // determine origination fee
        uint256 proratedOriginationFee = 0;
        if (fees.originationFee != 0) {
            proratedOriginationFee = FeesLib._calculateOriginationFee(fees,
amount, deadline, INTEREST_DENOMINATOR);
        }
```

It does calculate the `remainingTimeToLive` which is basically the deadline minus the timestamp inside the `_calculateOriginationFee` function.

As an example, let's say a 1 year remainingTimeToLive is calculated and paid for.

After 6 months, both the borrower and lender decide to refinance with still a 6-month duration on the initial credit, which they paid the originationFee from. They refinance with again, a deadline of 1 year.

First, the new deadline gets updated to account for a year from now:

```
  if (deadline != 0) {
            if (deadline <= block.timestamp) {
                revert InvalidDeadline();
            }
            credits[tokenId].deadline = deadline;
            emit SetDeadline(tokenId, deadline);
        }
```

Second, the new origination fee gets calculated:

```
  uint256 proratedOriginationFee = 0;
        if (fees.originationFee != 0) {
            uint256 remainingTimeToLive = credits[tokenId].deadline -
block.timestamp;
            proratedOriginationFee = FeesLib._calculateOriginationFee(
                fees,
                amount,
                block.timestamp + remainingTimeToLive,
                INTEREST_DENOMINATOR  );
```

It does calculate the `remainingTimeToLive` again, which is 1 year. Therefore the borrower will pay an extra 6 months of origination fee as they already paid for the initial year before hand instead of paying only for the 6 months extra that were not covered initially.

If the credit were to be refinanced again, it would happen repeatedly.

Impact:

`proratedOriginationFee` is overpaid on refinancing

Recommendation:

Do calculate the timestamp until it was paid on credit creation if a credit is refinanced before it expires.

Developer Response:

fixed at commit:

Line-Of-Credit-v2

https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/2a4dc4041e97c47d00f1f4f465a24cd915422b11
https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/3cd7734258afba2b6114f96430f27a6af91476e4
https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/08fbcbb27440ee0dc9c8674c8e3c8f95fe26f898

Vaults

https://github.com/credit-cooperative/Vaults/commit/768eca0b22fc3dbba9e7002097cb70b1959e599e

M-03 - Borrower address is not on sync between Line and Escrow

Severity: Medium Risk

Technical Details:

The Line of Credit contract has a setter function to update the address of the borrower in case the borrower needs it.

```
function updateBorrower(address newBorrower) public {
        {
        _onlyBorrower();
        if (newBorrower == address(0)) {
            revert InvalidAddress();
        }
        emit UpdateBorrower(borrower, newBorrower);
        borrower = newBorrower;
        nonce++;
    }
```

The issue is that the Line of Credit contract and the Escrow contract are initialized with the same borrower address from the factory. If the borrower address gets updated in the Line of Credit contract, then this should reflect in the Escrow contract.

Impact:

Borrower address is not on sync between Line and Escrow.

Recommendation:

Review if updating the borrower is a needed functionality inside the Line of Credit contract, if so, a function to update the borrower also needs to be created in the Escrow contract.

Developer Response:

fixed at commit:

https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/3d7496c62edbff420af53d4c36f6f20850816aa6
https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/7cad4b63011fde1fcbfc63644c3f21eba5187909

M-04 - Consider adding reentrancy protection in `sweep` from the `SpigotLine` contract

Severity: Medium Risk

Description:

`SpigotedLine.sol#sweep` function allows borrower to sweep the funds out.

```
    function sweep(address to, address token, uint256 amount) external {
        uint256 swept = SpigotedLineLib.sweep(
            to, token, amount, unusedTokens[token],
_updateStatus(_healthcheck()), borrower, admin, servicer
        );

        if (swept != 0) {
            unusedTokens[token] -= swept;
            emit ReservesChanged(token, -int256(swept), 1);
        }
    }
```

The function update state after the token transfer.

```
        if (
            status == LineLib.STATUS.REPAID
                && (
                    (msg.sender == borrower) || (msg.sender == admin && to
== borrower)
                        || (msg.sender == servicer && to == borrower)
                )
        ) {
            LineLib.sendOutTokenOrETH(token, to, amount);
            return amount;
        }
```

While the ETH token will not be used as collateral, certain `ERC777` tokens that have callback can re-enter the `sweep` function to remove more funds than the `unusedTokens[token]` amount.

Here is a list of liquid `ERC777` tokens:

VRA
https://etherscan.io/address/0xf411903cbc70a74d22900a5de66a2dda66507255

AMP
https://etherscan.io/address/0xff20817765cb7f73d4bde2e66e067e58d11095c2

LUKSO
https://etherscan.io/address/0xa8b919680258d369114910511cc87595aec0be6d

SKL
https://etherscan.io/address/0x00c83aecc790e8a4453e5dd3b0b4b3680501a7a7

imBTC
https://etherscan.io/address/0x3212b29e33587a00fb1c83346f5dbfa69a458923

CWEB
https://etherscan.io/address/0x505b5eda5e25a67e1c24a2bf1a527ed9eb88bf04

FLUX
https://etherscan.io/address/0x469eda64aed3a3ad6f868c44564291aa415cb1d9

More information:

https://mixbytes.io/blog/one-more-problem-with-erc777

Recommendation

Make this sweep function nonReentrant.

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/8cf3be1f2e1d9d035bcea7f35c7a60b57d3583ea

M-05 - Management fee is over estimated when the vaults are paused

Severity: Medium Risk

Technical Details:

Management fee is charged as a fee to manage the vault for the manager. This fee should not be charged whenever the vault is paused as the manager is not managing anything.

```
    // claim and mint shares for vault management fees
          _accrueManagementFee();
```

As the management fee calculation depends on `block.timestamp` to calculate the amount to be paid to the manager, the timestamp between the vaults are paused and unpaused need to be subtracted at the time of calling `_calculateProRataManagementFee()` after the vault get unpaused.

```
  function _calculateProRataManagementFee() internal view returns (uint256
proratedManagementFee) {
          uint256 timeSinceLastFee = block.timestamp -
lastManagementFeeTimestamp;
```

Impact:

Management fee will be overpaid after the vault is unpaused.

Recommendation:

The protocol needs to make and accrual of the vault management fee inside the pause function. Needs to track inside the `_calculateProRataManagementFee()` whether the protocol is paused, if so, skip the timestamp. After unpausing, accruals happen again without taking into account the time the vault was paused.

Developer Response: fixed at commit: https://github.com/credit-cooperative/Vaults/commit/1774320bd1e95328b2532e51a6bc30179584fdcd

M-06 - Spitgot lender repayment can be DOsed via 1 wei token transfer.

Severity: Medium Risk

Description:

The code `_repayLender` function has the logic below:

```
        uint256 beforeRepayment =
IERC20(beneficiary.creditToken).balanceOf(address(this));
        uint256 servicingFeeAmount =
FeesLib._calculateServicingFee(_getFees(), amount, BASE_DENOMINATOR);
        if (servicingFee != servicingFeeAmount) {revert
InsufficientFunds();}
        if (servicingFeeAmount > 0) {

IERC20(beneficiary.creditToken).safeTransfer(_getProtocolTreasury(),
servicingFee);}
        (bool success,) = beneficiary.poolAddress.call(functionData);
        if (!success) {revert OperatorFnCallFailed();}
        uint256 repaymentAmount = beforeRepayment -
IERC20(beneficiary.creditToken).balanceOf(address(this));
        if (repaymentAmount != amount + servicingFeeAmount) {revert
InsufficientFunds(); }
```

Note the check:

```
uint256 repaymentAmount = beforeRepayment -
IERC20(beneficiary.creditToken).balanceOf(address(this));
// ensure that not more than `amount` has left the contract

if (repaymentAmount != amount + servicingFeeAmount) {
    revert InsufficientFunds();
}
```

Suppose the input amount is 100 USDT, service fee is 10 UDST

Repayment amount is 100 USDT + 10 USDT = 110 USDT.

Any user can transfer 1 wei of the token to spigot contract, then the `beforeRepayment` is inflated by 1 wei.

Then the repayment amount will not be equal to 100 USDT and the transaction revert.

```
if (repaymentAmount != amount + servicingFeeAmount) {
    revert InsufficientFunds();
}
```

Recommendation:

Change the check to:

```
if (repaymentAmount < amount +  servicingFeeAmount) {
    revert InsufficientFunds();
}
```

Description: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/pull/110

M-07 - Unused approval is not reset after external call in LaasEscrow

Severity: Medium Risk

Description:

The function LaaSEscrow#drawdownAndIncreaseLiquidity :

calls LaasEscrow#LineLib._forceApprove, which forcefully gives the pool address approval.

After the execution flow is completed, even the approved amount is not fully consumed, the external pool will still hold some of the approval.

In case that the external pool is malicious or maliciously upgradeable, the logic below in the external pool contract can be used to transfer the funds from out from the Escrow.

```
  function stealFund(uint256 amount, address token, address recipient) {
    IERC20(token).transferFrom(address(LaasEscrow), recipient, amount)
  }
```

Recommendation

Forcefully reset the approval amount after the external call.

```
    LineLib._forceApprove(_getCreditToken(id), pool, amount);
    state._increaseLiquidity(liquidityPositions, index, lpToken,
increaseLiquidityData, pool);
    LineLib._forceApprove(_getCreditToken(id), pool, 0);
```

Developer Response:

Fixed at commit

https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/8dad78b6129fbf5bd86254c641d0578c344c8bba

and:

https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/34bf5f9fa2dc22de673e842a1c1355a00566dee8

M-08 - Increase liquidity and remove liquidity in LaasEscrow does not validate lp function signature

Severity: Medium Risk

Technical Details:

When liquidation position in LassEscrow, the code validates the calldata function signature.

```
        ILaaSEscrow.LiquidityPosition storage lp =
liquidityPositions[index];

        bytes4 func = bytes4(removeLiquidityData);

        if (lp.LpToken == address(0)) {
            // if LP position does not exist, revert
            revert ILaaSEscrow.NoLiquidityPosition();
        }

        if (func != lp.removeLiquidityFunc) {
            revert IEscrow.InvalidFunctionSelector();
        }
```

if the function signature does not match the lp.removeLiquidityFunc, transaction revert, the struct of ILaaSEscrow.LiquidityPosition is:

```
    /**
     * @param componentTokens the different tokens needed for the liquidity
position
     * @param componentTokenAmounts the amount of each token needed for the
LP position
     * @param LpToken the token we get after providing liquidity. will serve
as collateral in the escrow contract
     * @param liquidityProvisionData calldata for adding liquidity
     * @param removeLiquidityData the calldata for removing liquidity
     * @param liquidityProvisionFunc the function called to add liquidity
     * @param removeLiquidityFunc the function called to remove liquidity
     * @param apron is added on top of the mincratio. If this combined
threshold is breached, partial liquidation is allowed
     * @param isDelayOnLiquidityRemoval is there a delay when liquidty is
removed
     * @param isUniswapV3 is this strategy a uniswap v3 strategy?
     */
    struct LiquidityPosition {
        address[] componentTokens; // wont ever be more than 4 i think?
Curve pools
        uint256[] componentTokenAmounts;
        address LpToken;
        bytes liquidityProvisionData;
        bytes removeLiquidityData;
        bytes4 liquidityProvisionFunc;
        bytes4 removeLiquidityFunc;
        uint32 apron;
        bool isDelayOnLiquidityRemoval;
        bool isUniswapV3;
    }
```

as the comment said:

```
     * @param liquidityProvisionFunc the function called to add liquidity
     * @param removeLiquidityFunc the function called to remove liquidity
```

but when remove liquidity or increase liquidity, the code does not validate if the function signature is removeLiquidityFunc or
liquidityProvisionFunc

Impact:

When the code intend to increase or remove liqudiity, the caller can pass in a different function call to bypass the liquidityProvisionFunc and removeLiquidityFunc validation.

Recommendation:

In the _removeLiquidity and _increaseLiquidity function, add code to validate that function signature.

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/ac04d072e26b4a4b0f502795ef75ebac25a08383

M-09 - Incorrect WETH address used

Severity: Medium Risk

Technical Details:

WETH is hardcoded to the address in Ethereum mainnet inside the contracts, while credit plans deploying multi-chain. This is a mistake and WETH should not be a constant variable with the address in mainnet, it should be a variable where the address is updated on deployment in every chain.

```
        function getLatestAnswer(address token_) public virtual returns
(int256) {
        address token = token_ == LineLib.WETH ? Denominations.ETH : token_;
```

Impact:

Incorrect WETH address used

Recommendation:

Do not hardcode addresses in the system.

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/10a9109fe8b6c9031e12fb2c733275e80512042f

M-10 - Usage of a single DEX for token swaps is suboptimal

Severity: Medium Risk

Technical Details:

Some parts of the codebase use directly a curve v3 pool to swap tokens in spot. This is not a good practice given that you are settling down as curve as your only swapping option, and are subjected to worse rates than using an aggregator.

Impact:

Swaps will get worse rates and be subjected to MEV

Recommendation:

Consider using a DEX aggregator like 1inch, or 0X instead of swapping directly in curve

Developer Response:

Acknowledged. Will be fixed in v3

M-11 - Malicious borrower can use a blacklisted address to block lender from withdrawing funds early

Severity: Medium Risk

Technical Details:

A malicious borrower can use a blacklisted address to block lender from withdrawing funds early.

If the early withdrawal fee is set, the lender needs to pay the early withdrawal fee to the borrower.

However, the fee is transferred to the borrower directly.

```
  if (fee != 0) {
            IERC20(credits[tokenId].token).safeTransfer(borrower, fee); // NOTE: send fee from line to borrower
            emit EarlyWithdrawalFee(fee, msg.sender, borrower);
        }
```

Combing with the newly added code updateBorrower

```
    function updateBorrower(address newBorrower) public {
        _onlyBorrower();
        if (newBorrower == address(0)) {
            revert InvalidAddress();
        }
        emit UpdateBorrower(borrower, newBorrower);
        borrower = newBorrower;
        nonce++;
    }
```

The borrower can use a blacklisted address to not let the lender withdraw early.

Popular stable coins such as USDT or USDC are liquid and support blacklisting addresses.

https://dune.com/phabc/usdt---banned-addresses

Transferring any token to a blacklisted address will revert the whole transaction and thus block the lender from withdrawing funds.

Impact:

The lender is not able to withdraw early if the borrower address is updated to a blacklisted address.

Recommendation:

Let the borrower claim the early withdraw fee in a separate function and not transfer the fee out when the withdraw function is called.

```
        if (fee != 0) {
              earlyWithdrawalFee[tokenId] += fee;
              emit EarlyWithdrawalFee(fee, msg.sender, borrower);
          }
```

and

```
  function claimWithdrawalFee(uint256 tokenId, uint256 amount) {
    _onlyBorrower();
     earlyWithdrawalFee[tokenId] -= amount;
     IERC20(credits[tokenId].token).safeTransfer(msg.sender, amount);
 }
```

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/11f50e37caf7dc27a3b9cf4ace9cec7d611f3d14

M-12 - The `RefinanceCredit` function from LineOfCredit does not accure pending interest before changing the rate rate

Severity: Medium Risk

Technical Details:

The `LIneOfCredit` contract has a function refinanceCredit

where the interest rate of a loan tracked by token id can be changed by calling `_setRates`

```
    // allows dRate and fRate to be 0 for interest free credit positions
    _setRates(tokenId, dRate, fRate);
```

The function NATSPEC states that it accrues the interest before updating terms.

However, the pending interest is not accrued before updating the interest rate when the function refinance is called.

Impact:

Consider the initial interest is 10% weekly, the borrowed funds are 100 USDT:

1. 1 week goes by, and the owed pending interest 100 USDT * 10 % = 10 USDT
2. both, borrower and lender agree to refinance the terms and change the interest rate to 2%.
3. refinance position is called, the interest is accrued, the pending interested because 100 USDT * 2% = 2 USDT.

Because the previous 10% weekly interest is not settled, the lender loses part of the interest owed from borrower.

Recommendation:

Call _accrue before updating the interest rate

Developer Response: Fixed at commit:
https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/252eef06ccba0feb4ab382ff2059d4c392a28aca

M-13 - `reallocateFunds` logic in the StablecoinStrategy is broken

Severity: Medium Risk

Technical Details:

The `reallocateFunds` function intends to allow the owner to allocate a certain amount of funds inside the strategy to different whitelisted vaults.

```
 function reallocateFunds(address[] memory targets, uint256[] memory
amounts, bool[] memory isDeposit) external {
        _onlyLendingVaultOrOwner();
        _arrayLengthsMustMatch(targets, amounts, isDeposit);
        for (uint256 i = 0; i < targets.length; i++) {
            _notStakedUSDA(targets[i]);
            if (isDeposit[i]) {
                address depositedAsset = IERC4626(targets[i]).asset();
                if (depositedAsset != address(asset))
_exchange(address(asset), depositedAsset, amounts[i]);
                uint256 amountToDeposit =
IERC20(depositedAsset).balanceOf(address(this));
                uint256 sharesMinted =
IERC4626(targets[i]).deposit(amountToDeposit, address(this));
```

As can be seen, there is a logical issue inside the `if (isDeposit[i]) {` clause.

It first fetches the asset of the destination vault: `address depositedAsset = IERC4626(targets[i]).asset();` which can either be USDC, USDT, or DAI.

In case the asset is not the same as the current one in the strategy, it is swapped until there is enough to cover the amount specified as amountIn: `if (depositedAsset != address(asset)) _exchange(address(asset), depositedAsset, amounts[i]);`

The first problem is that it does not account in case there was already a balance of such asset inside the contract, making a swap that might not even have been needed.

The second problem is that it does deposit the entire amount regardless:

```
                uint256 amountToDeposit =
IERC20(depositedAsset).balanceOf(address(this));
                uint256 sharesMinted =
IERC4626(targets[i]).deposit(amountToDeposit, address(this));
```

Breaking the logic of allocating certain amounts to certain vaults. In case there would be an amount of such token before the swap, it would also be transferred.

It is even worse in the case where `depositedAsset = address(asset)` as it completely nullifies the amount specified as a function argument and directly deposits the entire balance to the vault:

```
                uint256 amountToDeposit =
IERC20(depositedAsset).balanceOf(address(this));
                uint256 sharesMinted =
IERC4626(targets[i]).deposit(amountToDeposit, address(this));
```

Impact:

There will be a misallocation of funds to unwanted vaults and DOS if multiple allocations for the same asset are processed. Additionally, the events are wrongly emitted for amounts deposited.

Recommendation:

Adopt the following code to account for both cases:

```
 function reallocateFunds(address[] memory targets, uint256[] memory
amounts, bool[] memory isDeposit) external {
        _onlyLendingVaultOrOwner();
        _arrayLengthsMustMatch(targets, amounts, isDeposit);

        for (uint256 i = 0; i < targets.length; i++) {
            _notStakedUSDA(targets[i]);

            if (isDeposit[i]) {
                address depositedAsset = IERC4626(targets[i]).asset();
-                // Swap the asset to the deposited asset if they are
different
-                if (depositedAsset != address(asset))
_exchange(address(asset), depositedAsset, amounts[i]);

+                if (depositedAsset != address(asset)){

+                 uint256 balanceBefore =
IERC20(depositedAsset).balanceOf(address(this));
+                // Swap the asset to the deposited asset if they are
different
+                  _exchange(address(asset), depositedAsset, amounts[i]);
+                 uint256 balanceAfter =
IERC20(depositedAsset).balanceOf(address(this));

+                 uint256 amountToDeposit = balanceAfter - balanceBefore;
+                uint256 sharesMinted =
IERC4626(targets[i]).deposit(amountToDeposit, address(this));

+                emit VaultDeposit(targets[i], depositedAsset,
amountToDeposit, sharesMinted);
+                } else{

-                uint256 amountToDeposit =
IERC20(depositedAsset).balanceOf(address(this));
-                uint256 sharesMinted =
IERC4626(targets[i]).deposit(amountToDeposit, address(this));
+                uint256 sharesMinted =
IERC4626(targets[i]).deposit(amounts[i], address(this));
                 emit VaultDeposit(targets[i], depositedAsset, amounts[i],
sharesMinted);
+            }
            } else {
```

Developer Response: fixed at commit: https://github.com/credit-cooperative/Vaults/commit/4c188dbf14dfba024ec3be406214195e141ab6ab

Low Risk

L-01 - Consider to de-couple `claimOwnerTokens`, `swap`, and `repayment` in `SpigotedLine`

Severity: Low Risk

Description:

The code involves claiming tokens from spigot and then swapping and repaying.

```
   function claimAndRepay(address claimToken, bytes calldata
zeroExTradeData) external returns (uint256) {
        _whileBorrowing();
        uint256 id = ids[0];
        Credit storage credit = credits[id];

        // only admin can call this function when _claimAndTrade is called
with claimToken != credit.token
        if (msg.sender != admin && claimToken != credit.token) {
            revert CallerAccessDenied();
        }

        // servicer can call this function when _claimAndTrade is called
with claimToken == credit.token
        if (!isServicer[msg.sender] && msg.sender != admin) {
            revert CallerAccessDenied();
        }
        _accrue(credit, id);

        uint256 newTokens = _claimAndTrade(claimToken, credit.token,
zeroExTradeData);
        uint256 repaid = newTokens + unusedTokens[credit.token];
```

This needs to call ISpigot(params.spigot).claimOwnerTokens.

```
   // @dev claim has to be called after we get balance
        // reverts if there are no tokens to claim
        uint256 claimed =
ISpigot(params.spigot).claimOwnerTokens(params.claimToken);
```

The problem is that the `claimOwnerToken` function can be called separately by the servicer role.

```
    function claimOwnerTokens(address token) external nonReentrant
onlyOwnerOrServicer returns (uint256) {
```

If the function `claimOwnerTokens` is called separately and not inside the transaction of `claimAndRepay` when the transaction `claimAndRepay` is executed, no token will be claimed.

Recommendation:

Consider de-coupling repay in `SpigotedLine` and restrict that only the `SpigotedLine` can call the function `claimOwnerTokens`.

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/0823f0dfd45e8d18ee18450ef26180d7f911cf73

L-02 - `mint()` should have access control

Severity: Low Risk

Technical Details:

The `mint()` function from the credit position token contract allows anyone to mint a CPT to any address without restrictions. While no other impact than having incorrect state has been found, it is advised to access control this function only to lines of credit.

```
function mint(address to, address line, bool isTransferRestricted) public
returns (uint256) {
        _tokenIds++;
        uint256 newItemId = _tokenIds;
        tokenToLine[newItemId] = line;
        _mint(to, newItemId);

        if (isTransferRestricted) {
            _isTransferRestricted[newItemId] = true;
        }
        return newItemId;
    }
```

Impact:

`mint()` should have access control, otherwise incorrect state can be given

Recommendation:

Only allow lines of credit to call this function.

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/512c8066bd9735098b88a9d046b2045749e7ab8f

L-03 - `accrueInterest()` can be DOSed if there are too many credit lines opened at the same time

Severity: Low Risk

Technical Details:

`accrueInterest()` loop through all the credit lines to accrue their interest:

```
function accrueInterest() external {
        uint256 len = ids.length;
        uint256 id;
        for (uint256 i; i < len; ++i) {
            id = ids[i];
            _accrue(credits[id], id);
        }
    }
```

If enough lines are open at the same time, the function could run out of gas.

Impact:

`accrueInterest()` can be DOSed if there are too many credit lines opened at the same time

Recommendation:

Add a maximum amount of credit lines possible to be open at the same time.

Developer Response:

We don't think this is worth fixing for 2 reasons:

- There would have to be a LOT of credit positions open on a line for this to be applicable
- Having an external function to call accrueInterest is a nice-to-have and not something we use for normal operations. Calling depositAndRepay, depositAndClose, close, etc. always accrues interest on a specific credit position before repayment.

Acknowledged

L-04 - `proposalCount` can be artificially inflated

Severity: Low Risk

Technical Details:

`proposalCount` gets increased every time one of the parties inside mutual consent "proposes" a certain msg.data to be accepted.

This can be increased multiple times by the exact same action. On party could call the same function multiple times, and all times `proposalCount` would get increased, while it shouldn't as it is not a new proposal.

While this doesn't have to have an impact that can be leveraged, it is not the right way the contract should behave. If it is the second time a function is being called with the same arguments, it should just return false and not emit events or increase the `proposalCount`

Impact:

`proposalCount` can be artificially inflated

Recommendation:
If it is the second time a function is being called with the same arguments, it should just return false and not emit events or increase the `proposalCount`

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/c6553e1b011e5ef0121a7bd46631bc462df8a800#diff-b78eb102810d9502c56b914d7a0e7fed82ac8b707c010c40b832337848d5b0db

L-05 - Nonce and deadline missing on "abort" and "setFees"

Severity: Low Risk

Technical Details:

Mutual consent actions require both parties to agree on a specific msg.data. This msg.data must change every time one of the key roles changes owner as the nonce should be updated.

When updating the borrower, the admin, or the servicer the nonce is increased as intended.

While this is checked in every function, this is not reflected in the abort and setFees functions.

Impact:

Nonce and deadline missing on "abort" and "setFees"

Recommendation:

Make the following changes to both the abort and the setFees functions:

```
-     function abort() external mutualConsent(admin, borrower) {
+    function abort(uint256 desiredNonce , uint256 deadline) external
mutualConsent(admin, borrower) {
+        if (desiredNonce != nonce) {
+            revert NonceMismatch();
+        }

+        if (deadline <= block.timestamp) {
+            revert InvalidDeadline();
+        }
        _updateStatus(LineLib.STATUS.ABORTED);
    }
```

Developer Response:

The nonce is updated so that borrower does not maliciously frontrun the lender (e.g. changing the minimum collateral ratio, spigot beneficiary allocations, etc.) before accepting an addCredit proposal. abort is called by mutual consent between admin and borrower so that the recovery functionality can be activated. This should not need nonce or deadline to be updated.

setFees is also called between mutual consent between admin and borrower so this suggestion does not apply there either.

acknowledged

L-06 - Incorrect permissions for registering contracts and whitelisting functions

Severity: Low Risk

Technical Details:

The `registerBorrowerContract()` and `whitelistFunction()` are able to be called by both the admin and the servicer, while it should only be callable by the admin. This is also specified in the NATSPEC of both functions.

```
  function registerBorrowerContract(address contractToRegister) external {
        _onlyAdminOrServicer();
        isContractRegistered[contractToRegister] = true;
    }


    /**
     * @notice - admin can whitelist certain functions on a registered
 borrower contract that can be called
     * @dev    - callable by admin or servicer
     * @param functionToRegister - the function signature to whitelist
     */
    function whitelistFunction(bytes4 functionToRegister) external {
        _onlyAdminOrServicer();
        whitelistedFunctions[functionToRegister] = true;
    }
```

Same happens with different permission with the `convertToClaimableTokens()` and `convertToCollateral()` functions, which allow the borrower and admin, while it should only allow the borrower, according to the NATSPEC.

Impact:

Incorrect permissions for registering contracts and whitelisting functions

Recommendation:

Do not allow the servicer role to call this functions.

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/5207a02b22e35fca58edaeaf5002806e214bfbb9

L-07 - if the default beneficiary is updated to an EOA account, the borrower can't call `resetBeneficiaries` and `updateBorrowerContractSplit`

Severity: Low Risk

Description:

The function `updateOwner` can be used to update the owner => default beneficiary.

```
    function updateOwner(address newOwner) public {
        if (msg.sender == _owner()) {
            return _updateDefaultBeneficiary(newOwner);
        }
```

The problem is that if the owner is updated to an EOA account or a smart contract that does not implement a function call such as `escrow()`,

the check _isSmartEscrow will revert directly when calling `address(IEscrowedLine(_owner()).escrow())`

```
   /// @notice - if smart escrow is being used, certain functions need to be
called from the escrow contract
    function _isSmartEscrow(address sender) internal returns (bool) {
        if (
            _borrower() == address(IEscrowedLine(_owner()).escrow())
                && sender ==
IEscrow(IEscrowedLine(_owner()).escrow()).borrower()
        ) {
            return true;
        }
```

if `_isSmartEscrow` reverts, even the debt is fully repaid, the borrower / admin cannot call resetBeneficiaries or updateBorrowerContractSplit

Recommendation:

Remove the check `_isSmartEscrow`

Developer Response: Acknowledged

L-08 - Proposals are kept open even when borrower gets updated

Severity: Low Risk

Technical Details:

Given that the nature of executing actions requires of a mutual consent between the lender and the borrower, whenever the borrower address is changed, the open proposals should automatically close given that are not directed to the current borrower.

```
function updateBorrower(address newBorrower) public {
        {
        _onlyBorrower();
        if (newBorrower == address(0)) {
            revert InvalidAddress();
        }
        emit UpdateBorrower(borrower, newBorrower);
        borrower = newBorrower;
        nonce++;
    }
```

Impact:

Proposals are kept open even when borrower gets updated, which could allow proposals to be executed if borrower reclaims the role inside the deadline.

Recommendation:

Either require for all proposals to be closed, or close them directly.

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/56312aee338f4e21900c2be533d6c83ade39bd5d

L-09 - Overall griefing capacity between lenders and borrowers

Severity: Low Risk

Technical Details:

When adding credit via `addCredit()` either the lender or the borrower can call this function async. Whoever calls the function last will be able to pull the required funds from the other party because the lender needs to transfer the funds and the borrower pay origination fee.

`LineLib.receiveTokenOrETH(token, lender, amount);`

Any party can grief the other party by removing the approval of tokens so that the transfer fails. Additionally, either party can also be griefed by revoking the consent as soon as the other party tries to execute such action.

Impact:

Overall griefing capacity between lenders and borrowers

Recommendation:

While it is a minimal problem, this is just a gotcha of the architecture, and while the lenders and borrowers are trusted and act in good faith, it should not happen. We would not propose to do a direct fix because things would get more complex. We think it is wise to add it as a comment in the code or add it to a list of known issues.

Developer Response: acknowledged

L-10 - Vault managers can be griefed on executions

Severity: Low Risk

Technical Details:

The vault has a two-step process for deposits and redeems. The user requests the deposit/redeem, and the vault manager accepts or denies such requests.

Given that the manager is the middle men, acting as a "keeper", he is subjected to griefing. A malicious depositor can be monitoring whenever the manager calls any of the functions to process the requests and "front-run" the manager by cancelling the open request, making the manager's transaction fail.

Impact:

Vault managers can be griefed on executions

Recommendation:

This would be easily solvable by executing all manager operations via a private mempool and/or not allowing to cancel a request after certain amount of time has passed.

Developer Response: Acknowledged

L-11 - Do not hardcode slippage while swapping

Severity: Low Risk

Technical Details:

The `swapThroughCRVPool` function allows the owner to swap WETH for stETH or vice versa using the corresponding curve pool.

Currently, the slippage is hardcoded to be a maximum of a 0.5% of slippage. While it should work, this will most likely be an inaccurate amount in most cases. It is a better option to calculate the slippage on the moment, there are times where 0.1% slippage might be possible, or other times that 0.5% might be too little, depending on market movements and liquidity.

Impact:

The slippage to swap in the WETH to stETH curve pool is slightly inaccurate.

Recommendation:

Allow to pass the slippage parameter on the spot, without hardcoding it to a certain value:

```
- function swapThroughCRVPool(uint256 amount, bool wethToSteth) external {
+ function swapThroughCRVPool(uint256 amount, bool wethToSteth, uint256 
amountOutMin) external {

-  uint256 stETHOut = crvPool.exchange{value: amount}(0, 1, amount, 
amount.mulDiv(MAX_SLIPPAGE, DENOM));
+  uint256 stETHOut = crvPool.exchange{value: amount}(0, 1, amount, 
amountOutMin);
```

Developer Response:

fixed at commit: https://github.com/credit-cooperative/Vaults/commit/cd43cdcc99e2e341c7fac0d2e3fa881ad925adbb

L-12 - `WithdrawAll()` can fail to 0 asset withdrawals

Severity: Low Risk

Technical Details:

`WithdrawAll()` loops through all the vaults whitelisted to withdraw the entirety of the position help by the strategy address. This can cause a problem if one of the vaults, recently created or empty, returns 0 as the `maxWithdrawable` amount. As it is common that certain tokens revert on 0 value transfers and could make the entire function fail.

```
function withdrawAll() public virtual {
        _onlyLendingVaultOrOwner();

        // Withdraw all assets from vaults
        for (uint256 i = 0; i < vaults.length; i++) {
            address asset = IERC4626(vaults[i]).asset();
            uint256 maxWithdrawable =
IERC4626(vaults[i]).maxWithdraw(address(this));
            uint256 sharesBurned =
IERC4626(vaults[i]).withdraw(maxWithdrawable, lendingVault, address(this));
            emit VaultWithdraw(vaults[i], asset, maxWithdrawable,
sharesBurned);
        }

        // Transfer Strategy assets to LendingVault
        uint256 withdrawableAssets = asset.balanceOf(address(this));
        asset.safeTransfer(lendingVault, withdrawableAssets);
        emit StrategyWithdraw(msg.sender, address(asset),
withdrawableAssets);
        ILendingVault(lendingVault).claimPerformanceFee();
    }
```

Impact:

A minor DOS can be given when the withdrawal amount of a certain vault is 0.

Recommendation:

```
function withdrawAll() public virtual {
        _onlyLendingVaultOrOwner();

        // Withdraw all assets from vaults
        for (uint256 i = 0; i < vaults.length; i++) {
            address asset = IERC4626(vaults[i]).asset();
            uint256 maxWithdrawable =
IERC4626(vaults[i]).maxWithdraw(address(this));
+         if ( maxWithdrawable > 0){
            uint256 sharesBurned =
IERC4626(vaults[i]).withdraw(maxWithdrawable, lendingVault, address(this));
            emit VaultWithdraw(vaults[i], asset, maxWithdrawable,
sharesBurned);
+             }
        }

        // Transfer Strategy assets to LendingVault
        uint256 withdrawableAssets = asset.balanceOf(address(this));
        asset.safeTransfer(lendingVault, withdrawableAssets);
        emit StrategyWithdraw(msg.sender, address(asset),
withdrawableAssets);
        ILendingVault(lendingVault).claimPerformanceFee();
    }
```

Developer Response:

Fixed at commit:
https://github.com/credit-cooperative/Vaults/commit/5035e790b9e6ecef27a23486f1c00d249a6e9980

L-13 - Consider whitelisting the token0 + token1 + fee tier for uniswap v3 nft in the `Escrow` contract.

Severity: Low Risk

Description:

The admin can enable a token pair as a collateral.

```
    function enableTokenPairAsCollateral(address token0, address token1)
external {
        _onlyAdmin();
        state.enablePairAsCollateral(token0, token1);

        emit EnablePairAsCollateral(token0, token1);
    }
```

Later the token nft id can be enabled if the check state.isValidPair is passed.

```
    /**
     * see Escrow.enablePosition
     */
    function enablePosition(
        Univ3State storage self,
        EscrowState storage state,
        address token0,
        address token1,
        uint256 tokenId,
        address oracle
    ) external returns (bool) {
        if (!state.isValidPair(token0, token1)) revert InvalidCollateral();
        return _enablePosition(self, tokenId, oracle);
    }
```

The problem is that knowing token0 and token1 is not sufficient to know which uniswap v3 pool the token id is in.

The uniswap v3 pool is created from token0 + token1 + fee tier.

```
    /// @inheritdoc IUniswapV3Factory
    function createPool(
        address tokenA,
        address tokenB,
        uint24 fee
    ) external override noDelegateCall returns (address pool) {
        require(tokenA != tokenB);
        (address token0, address token1) = tokenA < tokenB ? (tokenA,
tokenB) : (tokenB, tokenA);
        require(token0 != address(0));
        int24 tickSpacing = feeAmountTickSpacing[fee];
        require(tickSpacing != 0);
        require(getPool[token0][token1][fee] == address(0));
        pool = deploy(address(this), token0, token1, fee, tickSpacing);
        getPool[token0][token1][fee] = pool;
        // populate mapping in the reverse direction, deliberate choice to
avoid the cost of comparing addresses
        getPool[token1][token0][fee] = pool;
        emit PoolCreated(token0, token1, fee, tickSpacing, pool);
    }
```

As an example:

https://app.uniswap.org/explore/tokens/ethereum/0xa0b86991c6218b36c1d19d4a2e9eb0c

The token USDC has

USDC / ETH Pool with 0.05% fee tier and has 147M TVL

USDC / ETH Pool with 0.3% fee tier and has 66.4M TVL

USDC / ETH Pool with 0.01% fee tier and only has the 5.5M TVL

If a token pair has multiple pool with different fee tier, certain pool with fee tier may have very low liquidity and the protocol should avoid let user deposit fund into those low liquidity pool and mint nft as collateral.

Otherwise a single swap in the low liquidity pool can change the nft valuation a lot and impact the debt health.

Recommendation:

```
    function enableTokenPairAsCollateral(address token0, address token1,
feeTier) external {
        _onlyAdmin();
        state.enablePairAsCollateral(token0, token1, feeTier);

        emit EnablePairAsCollateral(token0, token1,  feeTier);
    }
```

Developer Response: fix: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/96d57c7c39e1bcd1dd20d652595971069fba3bd9

L-14 - If the `accrueManagementFee` function is not called for a prolonged period of time, accrueManagementFee can revert.

Severity: Low Risk

Description

The management fee is accrued by calling accrueManagementFee

```
    function accrueManagementFee() external nonReentrant returns (uint256) {
        _onlyVaultManagerOrOperator();
        uint256 newShares = _accrueManagementFee();
        return newShares;
    }

    function _calculateManagementFeeInflation(uint256 _feePercentage)
        internal
        view
        returns (uint256 newManagerShares)
    {
        newManagerShares =
_feePercentage.mulDiv(ERC20(lendingVault).totalSupply(), baseUnit -
_feePercentage);
    }

    function _calculateProRataManagementFee() internal view returns (uint256
proratedManagementFee) {
        uint256 timeSinceLastFee = block.timestamp -
lastManagementFeeTimestamp;
        uint256 managementFeeInBaseUnit = baseUnit.mulDiv(fees.management,
10000);
        proratedManagementFee =
timeSinceLastFee.mulDiv(managementFeeInBaseUnit, ONE_YEAR_IN_SECONDS);
        return proratedManagementFee;
    }

    /**
     * @notice  - Accrues management fees.
     * @return  - Amount of new shares minted.
     */
    function _accrueManagementFee() internal returns (uint256) {
        uint256 newVaultShares;
        if (fees.management > 0) {
            uint256 inflationFeePercentage =
_calculateProRataManagementFee();

            newVaultShares =
_calculateManagementFeeInflation(inflationFeePercentage);
```

The fee is accrued based on timestamp elapse.

If we consider the setting where the fee is set to 100%, a year elapses and the `accrueManagementFee` is not called, then calling `accrueManagementFee` will revert because when `_feePercentage >= baseUnit`, the transaction will revert in a division by zero or an underflow error.

```
    function _calculateManagementFeeInflation(uint256 _feePercentage)
        internal
        view
        returns (uint256 newManagerShares)
    {
        newManagerShares =
_feePercentage.mulDiv(ERC20(lendingVault).totalSupply(), baseUnit -
_feePercentage);
    }
```

Below is a simulation in python:

```
time_since_last_fee = 86400

base_unit = 10 ** 6

manage_fee = 10000

manage_fee_base_unit = base_unit * manage_fee / 10000

one_year_in_seconds = 31557600

proratedManagementFee = time_since_last_fee * manage_fee_base_unit /
one_year_in_seconds

print(proratedManagementFee)

print(base_unit)

total_supply  = 1000000 * base_unit

newManagerShares = proratedManagementFee * total_supply / (base_unit -
proratedManagementFee)

print(newManagerShares)
```

`one_year_in_seconds` is 365.25 days * 86400, and we assume the token has 6 decimals and the base unit is 10**6.

Recommendation:

Not calling `accrueManagementFee` for a long time is a rare case, regardless the protocol must consider calling this function in a timely manner.

Developer Response: fixed at commit: https://github.com/credit-cooperative/Vaults/pull/73

L-15 - Code comment and implementation is out of sync when swapping credit positions

Severity: Low Risk

Description

CreditStragety#IERC20(buyToken).transferFrom(buyer, address(this), buyTokenAmount) transfers the token from buyer to `address(this)` when swapping credit positions.

However, the comment said:

```
    // buyer pays for Credit Position Token -- transfer is to 7540 holding
contract instead of this
```

I think we are referring to "7540 holding contract" - the lending vault contract.

Recommendation

Transfer the buyer funds to lending vault directly or remove the outdated comment.

Developer response:
Fixed at commit: https://github.com/credit-cooperative/Vaults/commit/94e11c6ad43335d28059ba5ba8bc8a35cdd64aee

L-16 - Borrower is not capable of repaying lender in Spigot.sol

Severity: Low Risk

Description

Borrower is not capable of repaying lender in Spigot.sol.

Spigot#_repayLender has the modifier Spigot#onlyAdminOrServicer

```
modifier onlyAdminOrServicer() {
    if (msg.sender != _admin() && msg.sender != _servicer()) revert
CallerAccessDenied();
    _;
}
```

which makes the borrower not able to repay the lender in spigot.

Recommendation

Allow borrower to repay the lender in Spigot as well.

Protocol Response

Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/pull/101/commits

L-17 - Whilelisted smart contract and method in Escrow contract cannot be removed.

Severity: Low Risk

Technical Details:

Once the contract is registered and the method are whitelisted, the admin or borrower can operate to perform arbitrary call.

```
       function registerBorrowerContract(address contractToRegister)
external {
        _onlyAdminOrServicer();
        isContractRegistered[contractToRegister] = true;
    }

    /**
     * @notice - admin can whitelist certain functions on a registered
borrower contract that can be called
     * @dev    - callable by admin or servicer
     * @param functionToRegister - the function signature to whitelist
     */
    function whitelistFunction(bytes4 functionToRegister) external {
        _onlyAdminOrServicer();
        whitelistedFunctions[functionToRegister] = true;
    }

    /**
     * @notice - borrower and admin can call functions on registered
contracts
     * @dev    - callable by borrower or admin
     * @param targetContract - the address of the contract to call
     * @param data           - the calldata for the function to call
     */
    function operate(address targetContract, bytes calldata data) external {
        _onlyBorrowerOrAdmin();
        state.operate(isContractRegistered, whitelistedFunctions,
targetContract, data);
        _sync();
        _checkCollateralRatio();
    }
```

However, once the contract is registered and method is whitelisted, the contract and method cannot be removed.

Impact:

once the contract is registered and method is whitelisted, the contract and method cannot be removed even the contract and method are deprecated and should no longer to be called by borrower or admin.

Recommendation:

I understand that the protocol have concern about the init code size.

I think the change can be

```
    function registerBorrowerContract(address contractToRegister, bool
state) external {
        _onlyAdminOrServicer();
        isContractRegistered[contractToRegister] = state;
    }

    /**
     * @notice - admin can whitelist certain functions on a registered
borrower contract that can be called
     * @dev    - callable by admin or servicer
     * @param functionToRegister - the function signature to whitelist
     */
    function whitelistFunction(bytes4 functionToRegister, bool state)
external {
        _onlyAdminOrServicer();
        whitelistedFunctions[functionToRegister] = state;
    }
```

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/dbb3a30971295d3f934fb48f2085425c3341c773

L-18 - Chainlink oracle does not check if the sequencer is down in L2 network.

Severity: Low Risk

Technical Details:

The `getLatestAnswer` oracle function supports standard chainlink oracle query.

```
    function getLatestAnswer(address token_) public virtual returns (int256)
{
        address token = token_ == LineLib.WETH ? Denominations.ETH : token_;
        if (priceFeed[token_] == address(0)) {
            emit NoRoundData(token, "");
            return NULL_PRICE;
        }
        try AggregatorV3Interface(priceFeed[token_]).latestRoundData()
returns (
            uint80, /* uint80 roundID */
            int256 _price,
            uint256, /* uint256 roundStartTime */
            uint256 answerTimestamp, /* uint80 answeredInRound */
            uint80
        ) {
            // no price for asset if price is stale. Asset is toxic
            if (answerTimestamp == 0 || block.timestamp - answerTimestamp >
MAX_PRICE_LATENCY) {
                emit StalePrice(token, answerTimestamp);
                return NULL_PRICE;
            }
            if (_price <= NULL_PRICE) {
                emit NullPrice(token);
                return NULL_PRICE;
            }
```

However, the protocol intends to deploy the code in Arbtrium, which is an L2 network.

Handling Arbitrum outages

If the Arbitrum network becomes unavailable, the `ArbitrumValidator` contract continues to send messages to the L2 network through the delayed inbox on L1. This message stays there until the sequencer is back up again. When the sequencer comes back online after downtime, it processes all transactions from the delayed inbox before it accepts new transactions.

The message that signals when the sequencer is down will be processed before any new messages with transactions that require the sequencer to be operational.

But the Chainlink oracle integration does not check if the sequencer is down in L2 network.

Impact:

If the Arbitrum sequencer goes down, the protocol will allow users to continue to operate at the previous (stale) rates.

Recommendation:

Implement https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code to integrate with sequencer up time check.

Developer Response: Acknowledged

L-19 - Oracle returned value is not sufficiently validated.

Severity: Low Risk

Technical Details:

There are a lot of cases when the getLatestAnswer function can return `NULL_VALUE => return 0`.

Then the code that consumes the oracle value validates that the returned value is not 0.

Impact:

While some code validates that the returned value should not be 0,

```
            // if 4626 save the underlying token to use for oracle pricing
            deposit.asset = !is4626 ? token : abi.decode(tokenAddrBytes, 
(address));

            int256 price = IOracle(oracle).getLatestAnswer(deposit.asset);
            if (price <= 0) {
                return false;
            }
```

other places that consume the oracle to compute the debt or collateral worth do not validate if the oracle returns 0:

1. Collateral value computation does not validate oracle returned amount.

```
    collateralValue += CreditLib.calculateValue(o.getLatestAnswer(d.asset), 
deposit, d.assetDecimals);
```

2. Uniswap V3 Oracle does not validate if the latest answer is 0

```
        INonFungiblePositionManager.Position memory position =
INonFungiblePositionManager(nftPositionManager).positions(_tokenId);
        uint256 token0Price =
uint256(IOracle(oracle).getLatestAnswer(position.token0));
        uint256 token1Price =
uint256(IOracle(oracle).getLatestAnswer(position.token1));
```

3. OTC swap does not validate if the latest answer is 0.

```
 int256 tokenGoingOutUsdPrice =
IOracle(oracle).getLatestAnswer(otc.tokenGoingOut);
```

Recommendation:

Instead of validating if the return value is 0 every time when the getLatestAnswer is called, the code can validate if the oracle returns NULL_VALUE and revert in the getLatestAnswer function.

Developer Response: Acknowledged

L-20 - Claimable fund should be used to repay the debt first when the line of credit contract is subject to liquidation.

Severity: Low Risk

Technical Details:

In the current implementation, the deposit fund can be converted to claimable fund.

```
    function convertToClaimableTokens(EscrowState storage self, address
token, uint256 amount)
        external
        returns (bool)
    {
        if (self.deposited[token].amount < amount) {
            revert InvalidCollateral();
        }

        self.deposited[token].amount -= amount;
        self.tokensToClaim[token] += amount;
        return true;
    }
```

The claimer can claim the fund any time.

Impact:

Assume the claimer and borrower are same address.

1. Borrower deposit collateral.
2. Borrower convert a part of the collateral to claimable fund.
3. The line of credit is subject to liquidation and the escrow health factor too low.
4. Borrower (claimer) still claim the fund out.

Recommendation:

Claimable fund should be used to repay the debt first when the line of credit contract is subject to liquidation.

The protocol can disable the claim function when the line of credit is subject to liquidation and the escrow health factor too low.

```
    /**
     * @notice - claim claimable tokens
     * @dev    - callable by claimer
     * @param token  - the token to claim
     * @param amount - the amount to claim
     * @return       - true if successful
     */
    function claimTokens(address token, uint256 amount) external returns
(bool) {
        _checkCollateralRatio();
        if (_getLineStatus() == LineLib.STATUS.LIQUIDATABLE) {
            revert "should not claim token when health too low";
        }
        return state.claimTokens(token, claimer, amount);
    }
```

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/8cf49f460963468213b178b2acf35b2b13e77f48 and all claimable related code are removed.

L-21 - OTC swap should exclude the swap fee amount when computing the price impact

Severity: Low Risk

Technical Details:

A OTC swap can be executed in the Escrow contract.

```
    function otcSwap(OTCLib.OTC memory otc) external nonReentrant
mutualConsent(_getLineAdmin(), otc.counterparty) {
        state.otcCheck(otc);

        state.deposited[otc.tokenComingIn].amount +=
        // this is the amount we traded for minus the swap fee
        otcState.otcSwap(
            otc, ILineOfCredit(state.owner).getFees(), erc20Oracle,
ILineOfCredit(state.owner).protocolTreasury()
        );

        state.deposited[otc.tokenGoingOut].amount -=
otc.tokenGoingOutAmount;

        _checkCollateralRatio();
    }
```

If both, token in and token out are stable coins, the code just assumes that the stable coin value is strictly 1:1 by transferring in and out the same amount of token.

Note that in the line of code when computing the `tokenComingInAmountUsdValue`

```
     uint256 tokenComingInAmountUsdValue = CreditLib.calculateValue(
            tokenComingInUsdValue, otc.tokenComingInAmount + swapFeeAmount,
_getDecimals(otc.tokenComingIn)
        );
```

the code treats `otc.tokenComingInAmount + swapFeeAmount` as the token in amount.

However, the `swapFeeAmount` is charged separately and the fee goes to the protocol treasury, not the escrow contract.

Impact:

Including the swap fee to the token in amount would over-estimate the token in amount asset worth and impact price impact calculation.

Recommendation:

```
        uint256 tokenComingInAmountUsdValue = CreditLib.calculateValue(
            tokenComingInUsdValue, otc.tokenComingInAmount,
_getDecimals(otc.tokenComingIn)
        );
```

Developer Response: Acknowledged

L-22 - OTC swap assumes that all stablecoin are strictly 1:1 pegged

Severity: Low Risk

Technical Details:

A OTC swap can be executed in Escrow contract.

```
    function otcSwap(OTCLib.OTC memory otc) external nonReentrant
mutualConsent(_getLineAdmin(), otc.counterparty) {
        state.otcCheck(otc);

        state.deposited[otc.tokenComingIn].amount +=
        // this is the amount we traded for minus the swap fee
        otcState.otcSwap(
            otc, ILineOfCredit(state.owner).getFees(), erc20Oracle,
ILineOfCredit(state.owner).protocolTreasury()
        );

        state.deposited[otc.tokenGoingOut].amount -=
otc.tokenGoingOutAmount;

        _checkCollateralRatio();
    }
```

if both token in and token out are stable coin, the code just assume that the stable coin value is strictly 1:1 by transferring in and out the same amount of token.

```
   // see Escrow.otcSwap
    function otcSwap(
        OTCState storage self,
        OTC memory otc,
        ILineOfCredit.Fees memory fees,
        address oracle,
        address protocolTreasury
    ) external returns (uint256) {
        if (otc.expiration < block.timestamp) {
            revert ExpiredOrder();
        }

        // calculate swapFee for protocolTreasury
        uint256 swapFeeAmount = FeesLib._calculateSwapFee(fees,
otc.tokenComingInAmount, BASE_DENOMINATOR);
        if (self.stableCoinWhitelist[otc.tokenGoingOut] &&
self.stableCoinWhitelist[otc.tokenComingIn]) {
            if (otc.tokenGoingOutAmount != otc.tokenComingInAmount) {
                revert NotOnetoOne();
            }

            LineLib.sendOutTokenOrETH(otc.tokenGoingOut, otc.counterparty,
otc.tokenGoingOutAmount); // send stablecoin
            LineLib.receiveTokenOrETH(otc.tokenComingIn, otc.counterparty,
otc.tokenComingInAmount); // receive stablecoin
            LineLib.sendOutTokenOrETH(otc.tokenComingIn, protocolTreasury,
swapFeeAmount); // send swapFee to protocolTreasury
            return (otc.tokenComingInAmount - swapFeeAmount);
        }
```

However, it is not true that all stablecoin are strictly 1:1 pegged.

https://www.coindesk.com/markets/2024/01/03/usdc-stablecoin-momentarily-depegs-to-074-on-binance/

Even the most reliable stablecoin USDC depegged to 0.74 before.

Not to mention that the famous LUNC Luna stablecoin deppeg from 1 to 0.

Impact:

In case when the one of the stable coin depegs, the otc swap can be executed in suboptimal price.

Recommendation:

Just treat stablecoin like other asset and use oracle to compute the asset worth.

Developer Response:

acknowledged

L-23 - `earlyWithdrawalFee` when adding credit is not capped

Severity: Low Risk

Technical Details:

The `earlyWithdrawalFee` which is the fee the lender needs to pay if they withdraw funds before the deadline does not have a cap, allowing for that fee to be over 100%.

Impact:

`earlyWithdrawalFee` when adding credit is not capped

Recommendation:

Add a MAX percentage so that you can't charge penalties over certain amouns

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/2940695fe315823d425e9286f5cb67053a2782a8

L-24 - Balances are updated before token transfers (ANTI PATTERN)

Severity: Low Risk

Technical Details:
The `addCredit()` function updates key state changes that could potentially be leveraged and then pulls tokens from the lender. This is an ANTI PATTERN, as such states should be updated after the tokens are actually in the contract.

```
    function addCredit(
         uint256 desiredNonce,
         uint128 drate,
         uint128 frate,
         uint256 amount,
         address token,
         address lender,
         bool isRestricted,
         uint16 earlyWithdrawalFee,
         uint256 deadline
     ) external payable mutualConsent(lender, borrower) returns (uint256
 tokenId) {
         _whileActiveOrRepaid();

         if (desiredNonce != nonce) {
             revert NonceMismatch();
         }

         if (deadline <= block.timestamp) {
             revert InvalidDeadline();
         }

         tokenId = tokenContract.mint(lender, address(this), isRestricted);

         // create credit position
         _createCredit(tokenId, token, amount, isRestricted,
 earlyWithdrawalFee, deadline);

         // determine origination fee
         uint256 proratedOriginationFee = 0;
         if (fees.originationFee != 0) {
             proratedOriginationFee = FeesLib._calculateOriginationFee(fees,
 amount, deadline, INTEREST_DENOMINATOR);
         }

         _setRates(tokenId, drate, frate);

         if (status == LineLib.STATUS.REPAID) {
             _updateStatus(LineLib.STATUS.ACTIVE);
         }

         LineLib.receiveTokenOrETH(token, lender, amount);

         if (proratedOriginationFee != 0) {
             // send origination fees to line from borrower
             LineLib.receiveTokenOrETH(token, borrower,
 proratedOriginationFee);
             // send origination fees to protocol treasury
             LineLib.sendOutTokenOrETH(token, protocolTreasury(),
 proratedOriginationFee);
             emit TransferOriginationFee(proratedOriginationFee,
 protocolTreasury());
         }
```

Recommendation:

Pull the tokens from the lender before actually updating state.

```
    function addCredit(
        uint256 desiredNonce,
        uint128 drate,
        uint128 frate,
        uint256 amount,
        address token,
        address lender,
        bool isRestricted,
        uint16 earlyWithdrawalFee,
        uint256 deadline
    ) external payable mutualConsent(lender, borrower) returns (uint256
tokenId) {

        _whileActiveOrRepaid();

        if (desiredNonce != nonce) {
            revert NonceMismatch();
        }

        if (deadline <= block.timestamp) {
            revert InvalidDeadline();
        }

+        LineLib.receiveTokenOrETH(token, lender, amount);

        tokenId = tokenContract.mint(lender, address(this), isRestricted);

        // create credit position
        _createCredit(tokenId, token, amount, isRestricted,
earlyWithdrawalFee, deadline);

        // determine origination fee
        uint256 proratedOriginationFee = 0;
        if (fees.originationFee != 0) {
            proratedOriginationFee = FeesLib._calculateOriginationFee(fees,
amount, deadline, INTEREST_DENOMINATOR);
        }

        _setRates(tokenId, drate, frate);

        if (status == LineLib.STATUS.REPAID) {
            _updateStatus(LineLib.STATUS.ACTIVE);
        }

-        LineLib.receiveTokenOrETH(token, lender, amount);

        if (proratedOriginationFee != 0) {
            // send origination fees to line from borrower
            LineLib.receiveTokenOrETH(token, borrower,
proratedOriginationFee);
            // send origination fees to protocol treasury
            LineLib.sendOutTokenOrETH(token, protocolTreasury(),
proratedOriginationFee);
            emit TransferOriginationFee(proratedOriginationFee,
protocolTreasury());
        }
```

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/9d2ba31acf05bc8c3768130787e6f22c9cab50b9

L-25 - Incorrect symbol for credit position tokens

Severity: Low Risk

Technical Details:

The symbol in the CPT contract is supposed to be CPT which stands for credit position token, but it is LPT:

`constructor() ERC721("CreditPositionToken", "LPT") {}`

Impact:

Incorrect symbol for credit position tokens

Recommendation:

Update the symbol:

```diff
-   constructor() ERC721("CreditPositionToken", "LPT") {}
+   constructor() ERC721("CreditPositionToken", "CPT") {}
```

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/3e39a408a9df20979bdf4b140a84fb331d489306

L-26 - The `depositAndRepay` function in LineOfCredit should accrue interest first

Severity: Low Risk

Technical Details:

In LineOfCredit#depositAndRepay, the interest is accrued after the computation is done.

```
       if (amount > credit.principal + credit.interestAccrued +
servicingFeeAmount) {
            amount = credit.principal + credit.interestAccrued +
servicingFeeAmount;
        }

        _accrue(credit, id);
```

consider the case: `credit.principal`

```
credit.principal => 100 USDT

servicingFeeAmount => 1 USDT

credit.interestAccrued before accure => 10 USDT

User input is 200 USDT

The amount = 100 USDT + 1 USDT + 10 USDT = 111 USDT

Then the interest is accrued and the credit.interestAccrued becomes 20 USDT.

The true debt to be repaid is 100 UDST + 1 USDT + 20 USDT = 121 USDT,

Yet the amount is incorrectly trimmed to 111 USDT.
```

Impact:

Interest is not accrued when it should be accrued.

Recommendation:
Do update the code to the following:

```
    function depositAndRepay(uint256 amount) external payable {
        _whileBorrowing();
        uint256 id = ids[0];
        Credit storage credit = credits[id];

        // accure interest first
        _accrue(credit, id);

        uint256 servicingFeeAmount =
FeesLib._calculateServicingFeeFromAmount(fees, amount, BASE_DENOMINATOR);

        if (amount > credit.principal + credit.interestAccrued +
servicingFeeAmount) {
            amount = credit.principal + credit.interestAccrued +
servicingFeeAmount;
        }

        _repay(credit, id, amount, msg.sender);
    }
```

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/1c4028a4cfc5631e61fdde685f80970f65bff883

L-27 - LendingVault is not capable of calling the `depositIntoUSDA` function in USDAStrategy

Severity: Low Risk

Technical Details:

`USDAStragety.sol` is a liquid strategy.

The lendingVault contract has the function `depositIntoLiquidStrategy`:

```
 function depositIntoLiquidStrategy(uint256 amount) public {
        _onlyVaultManager();
        _assertVaultHasSufficientLiquidAssets(amount);
        liquidStrategy.deposit(amount);
    }

    /**
     * @notice  - Withdraws assets from the Liquid Strategy contract
     * @dev     - Only callable by the `manager` of the Credit Strategy
contract.
     * @param amount - Amount of assets to withdraw
     */
    function withdrawFromLiquidStrategy(uint256 amount) public {
        _onlyVaultManager();
        liquidStrategy.withdraw(amount);
    }
```

However, the function names in `USDAStragety.sol` are `depositIntoUSDA` and `withdrawFromUSDA`, not deposit and withdraw.

```
    function depositIntoUSDA(uint256 amountIn, uint256 amountOutMin, uint256
deadline) external {
        _onlyLendingVaultOrOwner();
        uint256 amountOut =
            transmuter.swapExactInput(amountIn, amountOutMin,
address(asset), USDA, address(this), deadline);
        emit StrategyExchange(address(asset), address(USDA), amountIn,
amountOut);

        uint256 sharesMinted = IERC4626(stUSDA).deposit(amountOut,
address(this));
        emit VaultDeposit(address(stUSDA), address(USDA), amountIn,
sharesMinted);
    }
```

The function is guarded by the `onlyLendingVaultOrOwner` modifier, so it is expected that both, the lending vault or the owner addresses can call this function.

The lending vault cannot call this function because it does not have the ability to call `depositIntoUSDA` and `WithdramFromUSDA` directly.

Impact:

The LendingVault is not capable of calling the `depositIntoUSDA` function.

Recommendation:

To fix this:

1. DepositIntoLiquidStrategy needs to be called from lendingVault and the target strategy is USDAStrategy.sol
2. The owner of USDAStrategy.sol needs to manually call depositIntoUSDA and withdrawFromUSDA.

It is recommended to only use the `onlyOwner` modifier in the functions `depositIntoUSDA` and `withdrawFromUSDA.`

Developer Response:

Code fixed at commit: https://github.com/credit-cooperative/Vaults/commit/60fb96d2d8e62036930b61f8e7fca24d241a26d5

Tests fixed at commit: https://github.com/credit-cooperative/Vaults/commit/6a7215455c07b8465a87c1e7fd2eb153858bc0fa

L-28 - An old pool will still have max approval after the pool address is changed in WETHStrategy contract

Severity: Low Risk

Technical Details:

The owner can set the new pool in WETHStrategy:

```
    function setCurvePool(address _newPool) external onlyOwner {
        crvPool = ICurvePoolPayable(_newPool);
        IERC20(steth).forceApprove(_newPool, type(uint256).max); // set
maximum allowance for steth
        emit CurvePoolSet(_newPool);
    }
```

However, the old approval is never cleared.

Impact:

The old and outdated pool will still have the max approval, and in the worst case, the old and outdated pool can still transfer funds out of the `WETHVault`.

Recommendation:

Reset the pool approval to 0 before changing the pool address.

```
    function setCurvePool(address _newPool) external onlyOwner {
        IERC20(steth).forceApprove(crvPool, 0); // reset approval
        crvPool = ICurvePoolPayable(_newPool);
        IERC20(steth).forceApprove(_newPool, type(uint256).max); // set
maximum allowance for steth
        emit CurvePoolSet(_newPool);
    }
```

Developer Response: Fixed at commit: https://github.com/credit-cooperative/Vaults/commit/20ca288e392bfba61ab4cafd2fa26bee8683e41d

Informational

I-01 - Unused code across the repository

Severity: Informational

Technical Details:

The following instances are unused:

- `DiscreteDistribution` import in IEscrowFactory
- `CIRCUIT_BREAKER_TYPE` in Line of Credit and factory
- `IOracle` import, `ILineFactory` import, `IMutualConsent` import in `ILineOfCredit`
- `LineLib` import in SpigotedLine.
- `ILineFactory` import in Escrow.
- `OTCLib` import in LaasEscrow.
- `IUniswapV3Oracle` import in EscrowFactory
- `FeedRegistryInterface` in Oracle
- `ILineFactory` in Spigot
- `console` in CPT

Impact:

Un-used code across the repository

Recommendation:

Remove the un-used code

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/2891fef63adf9e8bcc18d5a244c3c729f96b760b

I-02 - NATSPEC issues

Severity: Informational

Technical Details:

- 10000 bps is set to 1% while it should be 100% in OTCLib : `uint256 constant BASE_DENOMINATOR = 10000; // 10000 bps = 1%`
- Misspelling in the NATSPEC from the `close()` function in line of credit: `Requires that the position principal has already been repais in full` Should be repaid instead of repais.

Impact:

Informational

Recommendation:

Make the necessary corrections

Developer Response: fixed at commit: https://github.com/credit-cooperative/Line-Of-Credit-v2/commit/99ac9510508289117d31f40c097d17a84321a6c5

I-03 - CreditStrategySet event should take the parameter `creditStrategy` not `manager`

Severity: Informational

Technical Details: The `CreditStrategySet` event emits the contract address of the CreditStrategy, not the `manager` address of the CreditStrategy.

Impact: The term `manager` in the `CreditStrategySet` event is confusing to developers and users as it is mislabeled.

Recommendation: Rename `manager` in `event CreditStrategySet(address indexed manager);` to `creditStrategy`.

Developer Response:

Fixed at commit: https://github.com/credit-cooperative/Vaults/pull/63/commits/869fc83fa2759d625032e8d4cb03ca4ee80fb488

I-04 - No need to approve receiver address when swap credit position

Severity: Informational

Technical Details:
The approve function is not needed because the token is transferred out directly

```
IERC721(creditPositionToken).approve(receiver, tokenId);
IERC721(creditPositionToken).safeTransferFrom(address(this), receiver,
tokenId);
```

Recommendation:

remove

```
IERC721(creditPositionToken).approve(receiver, tokenId);
```

I-05 - SmartEscrow is not capable of handling native ETH

Severity: Informational

Technical Details:

Functions in `SmartEscrow` such as addCredit or refinanceCredit do not have the payable keywords:

```
   function addCredit(
        uint256 desiredNonce,
        uint128 drate,
        uint128 frate,
        uint256 amount,
        address token,
        address lender,
        bool isRestricted,
        uint16 earlyWithdrawalFee,
        uint256 deadline
    ) external {
        _onlyBorrower();

        ILineOfCredit(state.owner).addCredit(
            desiredNonce, drate, frate, amount, token, lender, isRestricted,
earlyWithdrawalFee, deadline
        );
    }
```

While line of credit contract is capable of handling and receiving the native ETH when adding credit or refinance credit.

```
   // send lender tokens to line
   LineLib.receiveTokenOrETH(token, lender, amount);
```

`SmartEscrow` users cannot lend native ETH out.

Impact:

The `SmartEscrow` users cannot lend native ETH out because all functions in SmartEscrow are missing payable keywords.

Recommendation:

Add the payable keyword or just wrap ETH to WETH and trade all tokens as ERC20 token to avoid the complexity of handling the native ETH case.

Developer Response:

Fixed at commit:
3c0f2c49b182cfa022c3c76023a31c3163af606

Gas Optimization

No issues found.

Disclaimer