CD SECURITY

AUDIT REPORT

Credifi

July 2025

Prepared by
ArnieSec
ZanyBonzy

Introduction

A time-boxed security review of the **Credifi** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Credifi**

The Credifi protocol offers unsecured loans. It is comprised of 3 main contracts and a library, these contracts include Credifi1155, CredifiERC20Adaptor, LibCredifiERC20Adaptor, and CredifiOracle

The oracle is responsible for providing the value of a token which is pegged to $1.00 for all supported tokens. The contract also utilizes decimal normalization to ensure that values returned by the oracle are always in 18 decimals. This allows the oracle to work with tokens of different decimals.

The Credifi1155 contract builds upon ERC1155 token standard. The contract adds whitelist functionality that ensures that only whitelisted entities can send or receive the tokens. The contract also integrates with the CredifiERC20Adaptor and adds convenience functions that allow for easy opening, paying, and closing of loans.

The CredifiERC20Adaptor allows the Credifi1155 tokens to integrate with the EVC and Euler vaults. This is done by receiving ERC1155 tokens and minting ERC20 CREDIT tokens to allow compatibility with the EVC and Euler vaults. Integrating with the EVC allows the protocol to implement a secure access control model by utilizing the reliability, robustness, and security of the EVC. The model ensures that loans are separated by sub-accounts in order to achieve loan isolation, and also enforces that users cannot directly manipulate vault positions for a given loan. The contract assumes the role of operator for a user and their sub-accounts. This allows the contract to act on behalf of the user in order to borrow tokens from a set of whitelisted vaults.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 43cd36a8f5537ffe197f982fe623ba011a5d1992**

Scope

The following folders were in scope of the audit:

- `src/*`

The following number of issues were found, categorized by their severity:

- Critical & High: 3 issues
- Medium: 5 issues
- Low & Info: 9 issues

Findings Summary

ID	Title	Severity	Status
[C-01]	Credifi1155s are permanently stuck in the adaptor after loan closure	Critical	Fixed
[H-01]	Incorrect function signature in `cleanupSubAccount`	High	Fixed
[H-02]	Direct call to `disableController` on vault is incorrect	High	Fixed
[M-01]	Incorrect account is checked for operator	Medium	Fixed
[M-02]	Users are limited to 156 sub accounts instead of 256	Medium	Fixed
[M-03]	Sub-Accounts may be DOS-ed temporarily	Medium	Fixed
[M-04]	Missing functionality to deposit and borrow making directly minted Credifi1155 to users unusable	Medium	Acknowledged
[M-05]	Calling `payoffAndCloseLoan` fails if debt still remains	Medium	Fixed
[L-01]	Setting up sub-account is done twice	Low	Fixed
[L-02]	Incorrect `INTEREST_RATE_PER_SECOND` constant	Low	Fixed
[L-03]	CredifiOracle ignores potential depeg risks allowing for arbitrage	Low	Acknowledged
[L-04]	Conflicting/Redundant token decimal check in oracle	Low	Fixed

ID	Title	Severity	Status
[L-05]	Pausing Credifi1155 also affects minting and burning	Low	Fixed
[L-06]	Credifi1155's metadata can be updated after a loan has been created	Low	Fixed
[L-07]	Unnecessary `transfer` and `transferFrom` overrides	Low	Fixed
[L-08]	Duplicate account setup during individual loan execution	Low	Fixed
[I-01]	Unused struct	Informational	Fixed

Detailed Findings

[C-01] Credifi1155s are permanently stuck in the adaptor after loan closure

Severity

Impact: High

Likelihood: High

Description

`payoffAndCloseLoan` is meant to close the loan and burn the associated Credifi1155 tokens.

```
    function payoffAndCloseLoan(address user, uint256 loanId, uint256
repayAmount)
        external
        onlyOwner
        validAddress(user)
    {
        require(credifiAdaptor != address(0), "CredifiERC20Adaptor not
set");

        ICredifiERC20Adaptor adaptor =
ICredifiERC20Adaptor(credifiAdaptor);

        // Close the loan (always burns associated tokens)
        // The owner (msg.sender) will provide funds for any outstanding
debt
        adaptor.closeIndividualLoan(user, loanId, repayAmount);

        // The adaptor will handle burning the CREDIT tokens and cleaning
up tracking
        // We don't need to burn the NFT here as it's already held by the
adaptor
    }
```

But in none of the associated functions is the Credifi1155 token burned. Only the minted credit tokens are burned upon vault redemption during collateral cleanup.

```
    function _withdrawCollateralAndCleanup(address user, IndividualLoan
storage loan) internal {
        // Use library to withdraw collateral
        uint256 redeemedCredits =
            LibCredifiERC20Adaptor.withdrawCollateral(evc,
loan.collateralVault, loan.subAccount, loan.creditAmount);

        // Clean up EVC access controls for the sub-account
        LibCredifiERC20Adaptor.cleanupSubAccount(evc, loan.subAccount,
loan.borrowVault);

        // Handle CREDIT tokens - always burn them
        if (redeemedCredits > 0) {
            // Burn operation: burn the CREDIT tokens directly (they're
already in this contract)
>           _burn(address(this), redeemedCredits);
        }

        // Remove token from user's deposited tokens
        _removeTokenFromUser(user, loan);

        emit IndividualLoanClosed(user, loan.borrowVault, loan.loanId,
loan.subAccount, loan.tokenId);
    }
```

As a result, Credifi1155 tokens are permanently stuck, irretrievably in the adaptor after loan closure.

Recommendations

Burn the tokens or return them to the user after the loan is closed.

[H-01] Incorrect function signature in `cleanupSubAccount`

Severity

Impact: Medium

Likelihood: High

Description

```
    function cleanupSubAccount(IEVC evc, address subAccount, address
borrowVault) external {
        // Remove borrow vault as controller
        bytes memory disableCalldata =
abi.encodeWithSignature("disableController(address)", subAccount);
```

The `disableCalldata` will include the incorrect function signature for `disableController` and thus the function will not be called. Below we can see what the correct function signature is and the function does not have an `address` parameter.

```
    function disableController() public virtual nonReentrant {
```

Recommendations

Use the correct function signature.

[H-02] Direct call to `disableController` on vault is incorrect

Severity

Impact: Medium

Likelihood: High

Description

In the function `cleanupSubAccount` the following logic is observed...

```
        bytes memory disableCalldata =
abi.encodeWithSignature("disableController(address)", subAccount);

        // Call the vault to disable itself as controller
        (bool success,) = borrowVault.call(disableCalldata);
```

The contract will attempt a direct call the vault. The result is that the function will attempt to `disableController` for the contract instead of the subAccount.

```
    function disableController() public virtual nonReentrant {
        address account = EVCAuthenticate();

        if (!vaultStorage.users[account].getOwed().isZero()) revert
E_OutstandingDebt();
```

```
        disableControllerInternal(account);
    }
```

If we look at the logic of `EVCAuthenticate` we can see that the msg.sender will be used since we are not utilizing an EVC call, instead we directly called the vault.

```
    function EVCAuthenticate() internal view virtual returns (address) {
        if (msg.sender == address(evc)) {
            (address onBehalfOfAccount,) =
evc.getCurrentOnBehalfOfAccount(address(0));

            return onBehalfOfAccount;
        }
        return msg.sender;
    }
```

Therefore the function will never correctly disable the vault as controller for the sub account.

Recommendations

Do not make a direct call to `disableController`, instead make the call via the EVC.

[M-01] Incorrect account is checked for operator

Severity

Impact: Low

Likelihood: High

Description

```
    function isAuthorizedOperator(address user) public view returns (bool)
{
        return evc.isAccountOperatorAuthorized(user, address(this));
    }
```

The above account will check if the current address is set as operator for a given address. However this check is incorrect in the following context.

```
        require(
            isAuthorizedOperator(user),
            "AUTH_REQUIRED" // CredifiERC20Adaptor must be authorized as
operator for user's sub-accounts
```

```
        );

        // Step 2: Enable the collateral vault for the sub-account (direct
call)
        // The EVC will authenticate this contract as an authorized
operator for the user's sub-accounts
        evc.enableCollateral(subAccount, collateralVault);

        // Step 3: Enable the borrow vault as controller for the sub-
account (direct call)
        evc.enableController(subAccount, borrowVault);
```

Here we check the user account in `isAuthorizedOperator` instead of the subAccount. Given that a user can set a different operator for each sub-account, this check is incorrect.

Recommendations

Call the function `isAuthorizedOperator` with the sub-account address instead.

[M-02] Users are limited to 156 sub accounts instead of 256

Severity

Impact: Medium

Likelihood: Medium

Description

The EVC allows users to have up to 256 sub-accounts, however users are unable to access all 256 sub-accounts when interacting with Credifi.

```
        uint256 subAccountId = userLoanCount[params.user] + 100;

        // Safety check: if this results in zero address or precompile
addresses, find a safe ID
        address subAccount =
LibCredifiERC20Adaptor.calculateSubAccount(params.user, subAccountId);
```

At the very minimum, the subAccountId will be == 100. This is a problem when we see the next snippet...

```
    function calculateSubAccount(address user, uint256 subAccountId)
internal pure returns (address) {
        require(subAccountId <= 255, "invalid subAccountId");
```

The snippet requires that the subAccountId <= 255. Since the lowest value for subAccountId is 100, only the values between 100 to 255 are allowed or else we revert above. Therefore the Credifi system is only compatible with 156 sub-accounts instead of the 256 that the EVC supports.

Additionally to that, after a user has done 156 loans, the calculation for his `subAccountId` will be == 256. This is a problem because of the logic of the `calculateSubAccount` function.

```
    function calculateSubAccount(address user, uint256 subAccountId)
internal pure returns (address) {
        require(subAccountId <= 255, "invalid subAccountId");
```

As we can see, if the `subAccountId` is 256 or greater, then the function will revert and thus the user will be unable to open a loan on Credifi since the function will continue to revert.

Recommendations

Allow users access to 256 sub-accounts by removing the addition of 100 in the subAccountId.

[M-03] Sub-Accounts may be DOS-ed temporarily

Severity

Impact: Medium

Likelihood: Medium

Description

In `LibCredifiERC20Adaptor` The function `cleanupSubAccount` utilizes try-catch statements when attempting to call `disableController` on a vault. However since the catch statement is a no op, the function allows for an unsuccessful call to this function to occur.

```
    function cleanupSubAccount(IEVC evc, address subAccount, address
borrowVault) external {
        // Remove borrow vault as controller
        bytes memory disableCalldata =
abi.encodeWithSignature("disableController(address)", subAccount);

        // Call the vault to disable itself as controller
        (bool success,) = borrowVault.call(disableCalldata);
        if (!success) {
            // If direct call fails, try through EVC
            try evc.call(borrowVault, subAccount, 0, disableCalldata) {
                // Success
            } catch {
                // Controller cleanup failed, but this is not critical for
loan closure
```

```
            }
        }
```

The comments correctly state that this is not important for loan closure however if the loan is closed, the user will be unable to disable the controller on his subAccount. This will cause his account to be DOSED since the EVC allows only 1 controller per account. Reasons why the call to `disableController` may fail are that the vault may include a pause guardian as a hook.

Recommendations

Consider ensuring that the call to `disableController` does not fail.

[M-04] Missing functionality to deposit and borrow making directly minted Credifi1155 to users unusable

Severity

Impact: High

Likelihood: Low

Description

Credifi1155:mint allows minting to a user, but after the token is minted to the user, it cannot be used to borrow or mint credit tokens

```
    function mint(address to, uint256 amount, bytes32 certificateHash,
bytes memory data)
        public
        onlyOwner
        returns (uint256)
    {
        require(to != address(0), "ZERO_ADDR");
        require(to != credifiAdaptor, "Cannot mint directly to adaptor");

        uint256 tokenId = _nextTokenId++;
        uint256 creationDate = block.timestamp;

        // Store token metadata
        TokenMetadata storage metadata = tokenMetadata[tokenId];
        metadata.creditAmount = amount;
        metadata.creationDate = creationDate;
        metadata.certificateHash = certificateHash;
        metadata.originalOwner = to;
        // _buffer_0 gets default value (all zeros) automatically

        _mint(to, tokenId, 1, data);

        emit TokenMinted(tokenId, certificateHash, to, amount,
```

```
creationDate);

        return tokenId;
    }
```

This is due to the missing `sendToVaultAndBorrow` and `sendToVaultAndBorrowOnBehalfOf` functionalities in CredifiERC20Adaptor.sol and functions to call them in Credifi1155.sol. These functions are expected, as can be seen in the interface ICredifiERC20Adaptor.sol.

```
    function sendToVaultAndBorrow(
        address collateralVault,
        address borrowVault,
        uint256 creditAmount,
        uint256 borrowAmount
    ) external;
    function sendToVaultAndBorrowOnBehalfOf(
        address user,
        address collateralVault,
        address borrowVault,
        uint256 creditAmount,
        uint256 borrowAmount
    ) external;
```

As a result, tokens directly minted to users are almost unusable when also factoring in the transfer restrictions.

Recommendations

Introduce the `sendToVaultAndBorrow` and `sendToVaultAndBorrowOnBehalfOf` functionalities in CredifiERC20Adaptor.sol and functions to call them in Credifi1155.sol. Otherwise, remove the `mint` functionality from Credifi1155.sol.

[M-05] Calling `payoffAndCloseLoan` fails if debt still remains

Severity

Impact: High

Likelihood: Low

Description

`payoffAndCloseLoan` is used to pay off any remaining debt and close the loan. It is expected that the caller, i.e the owner will provide funds for any outstanding debt.

```
    function payoffAndCloseLoan(address user, uint256 loanId, uint256
repayAmount)
        external
        onlyOwner
        validAddress(user)
    {
        require(credifiAdaptor != address(0), "CredifiERC20Adaptor not
set");

        ICredifiERC20Adaptor adaptor =
ICredifiERC20Adaptor(credifiAdaptor);

        // Close the loan (always burns associated tokens)
        // The owner (msg.sender) will provide funds for any outstanding
debt
        adaptor.closeIndividualLoan(user, loanId, repayAmount);

        // The adaptor will handle burning the CREDIT tokens and cleaning
up tracking
        // We don't need to burn the NFT here as it's already held by the
adaptor
    }
```

This is also backed up by the `closeIndividualLoan` function which works in two steps: first it handles any remaining debt and then withdraws collateral and cleanup.

```
    function closeIndividualLoan(address user, uint256 loanId, uint256
repayAmount)
        external
        nonReentrant
        validAddress(user)
    {
>>      require(msg.sender == address(credifi1155), "ONLY_CREDIFI"); //
Only Credifi1155 contract can call this function

        IndividualLoan storage loan = userLoans[user][loanId];
        require(loan.loanId == loanId, "Loan does not exist");

        // Handle any remaining debt first
        _handleRemainingDebt(user, loan, repayAmount);

        // Withdraw collateral and cleanup
        _withdrawCollateralAndCleanup(user, loan);
    }
```

When handling debt via `_handleRemainingDebt`, loan repayment is to be performed which transfers the borrow tokens from the payer, to the adaptor, and repays the debt. However, in this case, the payer is set, not as Credifi1155 contract owner (the origin caller as expected), but as `msg.sender` which is Credifi1155.sol.

```
    function _handleRemainingDebt(address user, IndividualLoan storage
loan, uint256 repayAmount) internal {
        // Convert 0 to max uint256 for backward compatibility with
payoffAndCloseLoan
        uint256 actualRepayAmount = repayAmount == 0 ? type(uint256).max :
repayAmount;
>>      _performLoanRepayment(user, loan, actualRepayAmount, msg.sender,
true);
    }
```

And as can be seen in `performLoanRepayment`, the borrowToken is attempted to be transferred from the payer to the adaptor contract. This call will fail since CredifiERC20Adaptor.sol is not approved to spend the borrow tokens, nor does the `payoffAndCloseLoan` function include logic to transfer the borrowToken from the caller owner to Credifi1155 contract.

```
    function performLoanRepayment(
        IEVC evc,
        IEulerVault borrowVault,
        IndividualLoan memory loan,
        uint256 repayAmount,
        address payer,
        bool requireFullRepayment
    ) external returns (uint256 actualRepayAmount, uint256 remainingDebt,
bool fullyRepaid) {
        uint256 currentDebt = borrowVault.debtOf(loan.subAccount);

        if (currentDebt > 0) {
            require(loan.active, "Loan is not active");

            IERC20 borrowToken = IERC20(borrowVault.asset());

            // If repayAmount is max uint256 or greater than debt, repay
full amount
            if (repayAmount == type(uint256).max || repayAmount >
currentDebt) {
                actualRepayAmount = currentDebt;
            } else {
                actualRepayAmount = repayAmount;
            }

            // Transfer repay tokens from payer to this contract
>>          borrowToken.safeTransferFrom(payer, address(this),
actualRepayAmount);

            // Approve vault to spend repay tokens
            borrowToken.approve(loan.borrowVault, actualRepayAmount);

            // Repay the debt through secure EVC call on behalf of the
sub-account
            bytes memory repayCalldata =
```

```
                abi.encodeWithSignature("repay(uint256,address)",
actualRepayAmount, loan.subAccount);
            secureVaultCall(evc, loan.borrowVault, address(this), 0,
repayCalldata);

            // Check remaining debt after repayment
            remainingDebt = borrowVault.debtOf(loan.subAccount);
            fullyRepaid = (remainingDebt == 0);

            // If full repayment is required (for loan closure), enforce
it
            if (requireFullRepayment) {
                require(fullyRepaid, "Loan still has outstanding debt
after repayment");
            }

            emit LoanRepaymentProcessed(
                loan.subAccount, loan.borrowVault, actualRepayAmount,
remainingDebt, fullyRepaid
            );
        } else {
        //...
```

As a result, the owner cannot directly repay and close the loan via the `payoffAndCloseLoan` function.

Recommendations

Update the _handleRemainingDebt function to set `Credifi1155.owner()` as payer instead of `msg.sender` or Use `safeTransferFrom` to transfer the borrowToken from the owner to Credifi1155.sol and approve CredifiERC20Adaptor.sol to spend the borrowToken.

[L-01] Setting up sub-account is done twice

Description

When calling `_executeIndividualLoanSetup` there is a call to `_setupSecureSubAccount` which calls `enableController`, and `enableCollateral`, however these calls are done again in a subsequent sub call later in the tx.

In `executeIndividualLoanSetup` we again attempt to setup secure sub account by calling `_setupSecureSubAccount`. This is not needed as the sub account has already been set up prior in the tx.

Recommendations

Consider removing unneeded call to `setupSecureSubAccount`

[L-02] Incorrect `INTEREST_RATE_PER_SECOND` constant

Severity

Impact: Low

Likelihood: Low

Description

`INTEREST_RATE_PER_SECOND` is calculated as `0.04 / 31536000 * 1e27` which is `1268391679350583460` not `1268391679350583552`. As a result, interest rate per second is slightly higher than expected.

```
    /// @dev Interest rate per second in 27 decimal precision (Euler
standard)
    /// Mathematically exact value for 4% APY: 0.04 / 31536000 * 1e27 =
1268391679350583552
    /// This precise constant avoids precision loss from integer division
    uint256 public constant INTEREST_RATE_PER_SECOND =
1268391679350583552;
```

Recommendations

Update the `INTEREST_RATE_PER_SECOND` constant to `1268391679350583460`

[L-03] CredifiOracle ignores potential depeg risks allowing for arbitrage

Severity

Impact: Medium

Likelihood: Low

Description

The protocol plans to work with stablecoins, USDC, Honey and Credifi's own erc20 token. `getQuote` returns 1 USD for all tokens. This is a potential issue in case of a token depeg. From major oracles, it can be observed that stablecoins, while pegged to 1 USD, are not always exactly priced at 1 USD. Since the oracle treats tokens to check as 1 USD, it can lead to arbitrage opportunities for users. In a more extreme case, if any of the stablecoins in use depeg, malicious users can take advantage of the price difference to profit, e.g (depositing a depeged, potentially worthless stablecoin to withdraw a more valuable one).

```
    function getQuote(uint256 inAmount, address base, address /* quote */
) external view returns (uint256 outAmount) {
        require(inAmount > 0, "CredifiOracle: Amount must be positive");
```

```
        // Get the base token decimals
        uint8 baseDecimals = _getTokenDecimals(base);

        // For CrediFi: all tokens are worth $1.00
        // Convert from base token decimals to 18-decimal unit of account
        if (baseDecimals < 18) {
            // Scale up: e.g., 100e6 USDC → 100e18 unit of account
            uint256 scaleFactor = 10 ** (18 - baseDecimals);
            outAmount = inAmount * scaleFactor;
        } else if (baseDecimals > 18) {
            // Scale down: e.g., 100e24 → 100e18 unit of account
            uint256 scaleFactor = 10 ** (baseDecimals - 18);
            outAmount = inAmount / scaleFactor;
        } else {
            // No scaling needed: already 18 decimals
            outAmount = inAmount;
        }

        return outAmount;
    }
```

Oracle is not actively in use in the contract, so the severity is lower.

Recommendations

A potential fix is using an actual battle-tested oracle for price comparisons, and/or introducing price bands. If the price is too far from 1 USD, the transaction should be reverted.

[L-04] Conflicting/Redundant token decimal check in oracle

Severity

Impact: Low

Likelihood: Low

Description

To check if a token is supported, the oracle holds the `supportsToken` function. This function checks if the token is a valid ERC20-like token by calling the `decimals()` function on the token. If the call to the `decimals` function fails, the function returns false.

```
    function supportsToken(address token) external view returns (bool
supported) {
        // Reject zero address and low addresses (precompiles, etc.)
        if (token == address(0) || uint160(token) <= 20) {
            return false;
        }
```

```
        // Check if address has code (is a contract)
        uint256 codeSize;
        assembly {
            codeSize := extcodesize(token)
        }
        if (codeSize == 0) {
            return false;
        }

        // Try to call decimals() to verify it's a valid ERC20-like token
        try IERC20TokenInfo(token).decimals() returns (uint8) {
            return true;
        } catch {
            return false;
        }
    }
```

But when getting price, in `getQuote`, the function calls `_getTokenDecimals` to get the decimals of the token and if it fails, it defaults to 18 decimals.

```
    function _getTokenDecimals(address token) internal view returns (uint8
tokenDecimals) {
        // Try to get decimals from the token directly
        try IERC20TokenInfo(token).decimals() returns (uint8 result) {
            return result;
        } catch {
            // Default to 18 decimals if call fails
            return 18;
        }
    }
```

This is conflicting and potentially redundant, as an external integration that calls `supportsToken` will fail if the token does not have a `decimals` function and therefore may not end up calling/using the `getQuote` function.

Recommendations

The catch check can be removed. If the `decimals` function fails, then the token is not supported and the `getQuote` function should fail as well.

[L-05] Pausing Credifi1155 also affects minting and burning

Severity

Impact: Low

Likelihood: Low

Description

Owner can pause token transfers. But due to the `ERC1155PausableUpgradeable` override in the `_update` function, minting and burning are also affected as they both rely on the `_update` function.

```
    /**
     * @dev Pause all token transfers (only owner)
     */
    function pause() public onlyOwner {
        _pause();
    }

    /**
     * @dev Unpause all token transfers (only owner)
     */
    function unpause() public onlyOwner {
        _unpause();
    }
```

```
    function _update(address from, address to, uint256[] memory ids,
uint256[] memory values)
        internal
        override(ERC1155Upgradeable, ERC1155PausableUpgradeable,
ERC1155SupplyUpgradeable)
    {
    ...
```

Recommendations

Leave as is if by design, or remove the pausing when minting and burning.

[L-06] Credifi1155's metadata can be updated after a loan has been created

Severity

Impact: Low

Likelihood: Low

Description

`updateMetadata` can be called to update a token's metadata.

```
    function updateMetadata(uint256 tokenId, uint256 newAmount, bytes32
newCertificateHash) public onlyOwner {
        require(exists(tokenId), "Token does not exist");

        tokenMetadata[tokenId].creditAmount = newAmount;
        tokenMetadata[tokenId].certificateHash = newCertificateHash;

        emit MetadataUpdated(tokenId, newCertificateHash, newAmount);
    }
```

The function however doesn't ensure that the token has not been used for a loan. As a result, the token's `creditAmount` and more importantly, the `certificateHash` will differ from the loan's `creditAmount` and `certificateHash`.

```
        IndividualLoan storage loan = userLoans[params.user][loanId];
//...
        loan.creditAmount = params.creditAmount;
        loan.timestamp = block.timestamp;
        loan.certificateHash = params.certificateHash;
//...
```

Recommendations

The function should check that the token has not been used for a loan before updating the metadata.

[L-07] Unnecessary `transfer` and `transferFrom` overrides

Severity

Impact: Low

Likelihood: Low

Description

`transfer` and `transferFrom` are overridden to prevent direct user transfers. But due to the way overrides work in Solidity, directly calling the `transfer` and `transferFrom` in ERC20Upgradeable.sol (i.e if they were not overridden) will still call the override `_update`, performing the needed whitelist validations.

```
    /**
     * @dev Override transfer to prevent direct user transfers
     * The _update function handles all whitelist validation
     */
    function transfer(address to, uint256 value) public virtual override
```

```
returns (bool) {
        return super.transfer(to, value);
    }

    /**
     * @dev Override transferFrom to prevent direct user transfers
     * The _update function handles all whitelist validation
     */
    function transferFrom(address from, address to, uint256 value) public
virtual override returns (bool) {
        return super.transferFrom(from, to, value);
    }
```

This makes the overrides redundant.

To prove this,

1. Add the function below to CredifiERC20Adaptor.sol

```
    function mint(address user, uint256 amount) external{
        _mint(user, amount);
    }
```

2. Comment out the aforementioned `transfer` and `transferFrom` overrides in CredifiERC20Adaptor.sol

```
    // function transfer(address to, uint256 value) public virtual
override returns (bool) {
    //     return super.transfer(to, value);
    // }

    // /**
    //  * @dev Override transferFrom to prevent direct user transfers
    //  * The _update function handles all whitelist validation
    //  */
    // function transferFrom(address from, address to, uint256 value)
public virtual override returns (bool) {
    //     return super.transferFrom(from, to, value);
    // }
```

3. Add the test below to CredifiERC20Adaptor.t.sol and run it

```
        function testTransferStillPerformsValidation() public {

        address user100 = makeAddr("user100");
        address user200 = makeAddr("user200");
```

```
        vm.startPrank(owner);

        adaptor.mint(user100, 100);
        vm.stopPrank();

        // Verify users cannot call loan management functions on the
adaptor directly
        vm.startPrank(user100);
        vm.expectRevert("VAULT_ONLY");
        adaptor.transfer(user200, 100);

        vm.stopPrank();
    }
```

It passes.

```
    ├─ [0] VM::startPrank(user100:
[0x1bF6364004bFC14ABc4A460280185165b3698661])
    │   └─ ← [Return]
    ├─ [0] VM::expectRevert(custom error 0xf28dceb3:

VAULT_ONLY)
│ └─ ← [Return]
├─ [7610] ERC1967Proxy::fallback(user200:
[0xC10569C4559F76D5db84f7F53F295058e9F87c8b], 100)
│ ├─ [7219] CredifiERC20Adaptor::transfer(user200:
[0xC10569C4559F76D5db84f7F53F295058e9F87c8b], 100) [delegatecall]
│ │ └─ ← [Revert] revert: VAULT_ONLY
│ └─ ← [Revert] revert: VAULT_ONLY
├─ [0] VM::stopPrank()
│ └─ ← [Return]
├─ storage changes:
│ @ 0x9b779b17422d0df92223018b32b4d1fa46e071723d6817e2486d003becc55f00: 2
→ 1
└─ ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.70ms
(1.10ms CPU time)
```

Recommendations

The functions can be safely removed.

[L-08] Duplicate account setup during individual loan execution

Severity

Impact: Low

Likelihood: High

Description

`_executeIndividualLoanSetup` calls `_setupSecureSubAccount` in CredifiERC20Adaptor.sol. The function also calls `executeIndividualLoanSetup` in LibCredifiERC20Adaptor.sol, which calls `setupSecureSubAccount`..

```
    function _executeIndividualLoanSetup(
        address user,
        address subAccount,
        address collateralVault,
        address borrowVault,
        uint256 creditAmount,
        uint256 borrowAmount
    ) internal {
        // Step 1: Transfer CREDIT tokens from contract to sub-account
        _internalTransfer(address(this), subAccount, creditAmount);

        // Step 2: Setup secure EVC access control for the sub-account
        // This establishes the CredifiERC20Adaptor as the primary
controller
        // and prevents direct user interaction with vaults for this loan
        _setupSecureSubAccount(user, subAccount, borrowVault,
collateralVault);

        // Step 3: Approve vault to spend tokens from sub-account
        // We need to bypass the normal approval restrictions for this
controlled operation
        // Use the internal _approve function with emitEvent=false to
avoid restrictions
        _approve(subAccount, collateralVault, creditAmount, false);

        // Step 4 & 5: Use library to execute loan setup
        LibCredifiERC20Adaptor.executeIndividualLoanSetup(
            evc, user, subAccount, collateralVault, borrowVault,
creditAmount, borrowAmount
        );

        // Note: Debt is tracked per-loan via the IndividualLoan struct
        // Event is emitted by calling function (IndividualLoanCreated)
    }
```

Both functions essentially do the same thing, checking that the contract is authorized operator and enabling the collateral and borrow vaults for the sub account.

Recommendations

Remove one of the function calls.

[I-01] Unused struct

Description

RepaymentContext struct is unused.

```
/**
 * @title RepaymentContext
 * @dev Struct to group repayment-related variables and reduce stack depth
 */
struct RepaymentContext {
    address borrowVault;
    address subAccount;
    uint256 currentDebt;
    uint256 remainingDebt;
    bool fullyRepaid;
}
```

Recommendations

It can be removed.