SB SECURITY

Credifi

Security Review

July 21, 2025

Contents

1. About SBSecurity

SBSecurity is a duo of skilled smart contract security researchers. Based on the audits conducted and numerous vulnerabilities reported, we strive to provide the absolute best security service and client satisfaction. While it's understood that 100% security and bug-free code cannot be guaranteed by anyone, we are committed to giving our utmost to provide the best possible outcome for you and your product.

Book a Security Review with us at sbsecurity.net or reach out on Twitter @Slavcheww.

2. Disclaimer

A smart contract security review can only show the presence of vulnerabilities **but not their absence**. Audits are a time, resource, and expertise-bound effort where skilled technicians evaluate the codebase and their dependencies using various techniques to find as many flaws as possible and suggest security-related improvements. We as a company stand behind our brand and the level of service that is provided but also recommend subsequent security reviews, on-chain monitoring, and high whitehat incentivization.

3. Risk classification

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

3.1. Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol's functionality.
- **Low** - funds are not at risk.

3.2. Likelihood

- **High** - almost **certain** to happen, easy to perform, or highly incentivized.
- **Medium** - only **conditionally possible**, but still relatively likely.
- **Low** - requires specific state or **little-to-no incentive**.

3.3. Action required for severity levels

- High - **Must** fix (before deployment if not already deployed).
- Medium - **Should** fix.
- Low - **Could** fix.

4. Executive Summary

Overview

Project	Credifi
Repository	Private
Commit Hash	00ba313cb217c1f7b1e9c85b971247a952510149
Resolution	ec3a18e79aa70f953280c8a1fd6e6fc7ce94941a
Timeline	July 9 - July 17, 2025

Scope

LibCredifiERC20Adaptor.sol
CredifiOracle.sol
CredifiERC20Adaptor.sol
Credifi1155.sol
ConstantRateIRM.sol

Issues Found

Critical Risk	1
High Risk	1
Medium Risk	2
Low/Info Risk	8

5. Findings

5.1. Critical severity

5.1.1.1. `payoffAndCloseLoan` is not assigning payer correctly

Severity: Critical Risk

Description: `payoffAndCloseLoan` mistakenly uses `Credifi1155` as a `payer`, which will make it impossible to close a loan.

`Credifi1155.payoffAndCloseLoan` is called by the contract owner → it calls `adaptor.closeIndividualLoan`, which then calls the internal `_handleRemainingDebt`.

As we see payer = msg.sender (`Credifi1155`)

```
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);//<---- msg.sender
}
```

Execution will revert with insufficient allowance in `performLoanRepayment` when tokens are being transferred. The reason is that `Credifi1155` doesn't approve the `Adaptor` and doesn't hold the tokens.

```
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);

        ... MORE CODE
```

Recommendation: In order to fix the issue w/o modifying the adaptor: 1. Pull the tokens from the owner of the `Credifi1155` file (caller of the `payoffAndCloseLoan`), 2. Approve the adaptor, 3. Retain the same logic.

Resolution: Fixed

5.2. High severity

5.2.1. Using pre-calculated collateral when closing loans is error-prone

Severity: High Risk

Description: 2 issues that will happen, when the amount collateral that's withdrawn differs from `creditAmount` which from the amount in `IndividualLoan`:

1. Adaptor mistakenly assumes only `creditAmount` is being withdrawn back from the Euler Vault, while in reality, there will be supply APY being generated.

```
function withdrawCollateral(IEVC evc, address collateralVault, address subAccount, uint256 creditAmount)
    external
    returns (uint256 redeemedCredits)
{
    IEulerVault collVault = IEulerVault(collateralVault);
    uint256 collateralShares = collVault.balanceOf(subAccount);

    if (collateralShares > 0) {
        // Redeem vault shares and send CREDIT tokens directly to this contract
        bytes memory withdrawCalldata =
            abi.encodeWithSignature("redeem(uint256,address,address)", collateralShares, address(this),
subAccount);
        secureVaultCall(evc, collateralVault, subAccount, 0, withdrawCalldata);

        // Get the amount of CREDIT tokens that were redeemed
        redeemedCredits = creditAmount; // Use the original credit amount for the loan
    }

    emit CollateralWithdrawn(subAccount, collateralVault, redeemedCredits);
    return redeemedCredits;
}
```

As we see, `redeemedCredits = creditAmount`, which is correct in terms of the Credifi tokens, initially minted when the loan was created, but any surplus that will be redeemed as a result of the APY will be ignored and will be locked in the `CredifiERC20Adaptor`.

2. When there is a partial liquidation, collateral withdrawn will be less than the creditAmount, but `Adaptor::_withdrawCollateralAndCleanup` will try to burn more than its balance, making the loan impossible to close.

```
function _withdrawCollateralAndCleanup(address user, IndividualLoan storage loan) internal {//OK
    // 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);
}
```

Recommendation: Use the return value of the redeem function, compare it to creditAmount, if it's more, burn the creditAmount and distribute the rest to users, otherwise, burn the returned amount only.

Resolution: Acknowledged

5.3. Medium severity

5.3.1. `withdrawCollateral` won't be possible if the utilization is high

Severity: Medium Risk

Description: When the utilization of the vault is high, there's a chance that not all the shares to be redeemable. In fact, this will prevent the `CredifiERC20Adaptor` from closing individual loans.

The root cause of the issue is the usage of `collVault.balanceOf` in the `LibCredifiERC20Adaptor` and the overall implementation of the `withdrawCollateral`:

```
function withdrawCollateral(IEVC evc, address collateralVault, address subAccount, uint256 creditAmount)
    external
    returns (uint256 redeemedCredits)
{
    IEulerVault collVault = IEulerVault(collateralVault);
    uint256 collateralShares = collVault.balanceOf(subAccount); //ISSUE: should use maxRedeem instead of
balanceOf

    if (collateralShares > 0) {
        // Redeem vault shares and send CREDIT tokens directly to this contract
        bytes memory withdrawCalldata =
            abi.encodeWithSignature("redeem(uint256,address,address)", collateralShares, address(this),
subAccount);
        secureVaultCall(evc, collateralVault, subAccount, 0, withdrawCalldata);

        // Get the amount of CREDIT tokens that were redeemed
        redeemedCredits = creditAmount; // Use the original credit amount for the loan
    }

    emit CollateralWithdrawn(subAccount, collateralVault, redeemedCredits);
    return redeemedCredits;
}
```

The problem is that `balanceOf` doesn't indicate how many shares can be withdrawn (the **actual** holdings of the EVault), but returns solely the balance of the sub account. `maxRedeem` instead is the function that returns a proper number, which is the max redeemable shares that the vault has.

```
function maxRedeem(address owner) public view virtual nonReentrantView returns (uint256) {
    VaultCache memory vaultCache = loadVault();
    if (isOperationDisabled(vaultStorage.hookedOps, OP_REDEEM)) return 0;

    Shares max = maxRedeemInternal(vaultCache, owner);
    // if shares round down to zero, redeem reverts with E_ZeroAssets
    return max.toAssetsDown(vaultCache).toUint() == 0 ? 0 : max.toUint();
}

function maxRedeemInternal(VaultCache memory vaultCache, address owner) private view returns (Shares) {
    Shares max = vaultStorage.users[owner].getBalance();

    // If account has borrows, withdrawal might be reverted by the controller during account status checks.
    // The vault has no way to verify or enforce the behaviour of the controller, which the account owner
    // has enabled. It will therefore assume that all of the assets would be withheld by the controller and
    // under-estimate the return amount to zero.
    // Integrators who handle borrowing should implement custom logic to work with the particular
controllers
    // they want to support.
    if (max.isZero() || hasAnyControllerEnabled(owner)) return Shares.wrap(0);

    Shares cash = vaultCache.cash.toSharesDown(vaultCache);
    max = max > cash ? cash : max;

    return max;
}
```

As we see the check in `maxRedeemInternal` caps the shares to the vault's `cash`, which is the amount of assets tracked as current holdings, i.e. amount available for withdrawal.

If we assume `maxRedeem` is being used instead of `balanceOf`, the issue still won't be fixed as the logic would mistakenly assume the entire `creditAmount` has been withdrawn, and the not withdrawn collateral will be locked in the vault, since the adaptor tokens have been burned.

As a summary, in vaults with high utilization, closing loans won't be possible, since redeem will be reverted due to insufficient cash in the `EVault`.

Recommendation: There should be a function that allows redeeming an arbitrary amount of collateral and proper `IndividualLoan` management. This way, both this and the `makeLoanPayment` functions can be paired to allow unwinding the position easily. Furthermore the order of operations, should also be reconsidered, since `maxRedeem` is only possible when the sub account has disabled all the controllers:

1. Repay debt

2. Disable controller

3. Redeem collateral

Resolution: Acknowledged

5.3.2. borrowVault is never disabled when cleanup is done

Severity: Medium Risk

Description: When closing a loan, `cleanupSubAccount()` must remove the `collateralVault` and `borrowVault` for the subAccount (aka the account prefix) because Euler does not allow multiple controllers to be activated simultaneously for a single prefix. (Multiple collaterals are acceptable, but that is not the issue here).

```
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
        }
    }

    // Disable collateral vaults
    address[] memory collaterals = evc.getCollaterals(subAccount);
    for (uint256 i = 0; i < collaterals.length; i++) {
        try evc.disableCollateral(subAccount, collaterals[i]) {
            // Success
        } catch {
            // Non-critical error - collateral will remain enabled but inactive
        }
    }
}
```

`cleanupSubAccount` tries to call `"disableController(address)"` on `borrowVault` itself with solidity's built-in low-level call and if that fails, it will try via `evc`. But the direct call will always fail because `borrowVault` has a `"disableController()"` with no parameters and this `"disableController(address)"` function will not be found, then it will try `evc.call()`, which again performs a low-level call to the specified target contract (1st parameter), but also checks subAccount for authentication.

It will check the subAccount and then invoke `callWithContextInternal()`, where it will finally do the same `borrowVault.call(disableCalldata);` and will fail again, since `borrowVault` does not have this function defined.

```
function callWithContextInternal(
    address targetContract,
    address onBehalfOfAccount,
    uint256 value,
    bytes calldata data
) internal virtual returns (bool success, bytes memory result) {
    if (value == type(uint256).max) {
        value = address(this).balance;
    } else if (value > address(this).balance) {
        revert EVC_InvalidValue();
    }

    EC contextCache = executionContext;
    address msgSender = _msgSender();

    // set the onBehalfOfAccount in the execution context for the duration of the external call.
    // considering that the operatorAuthenticated is only meant to be observable by external
    // contracts, it is sufficient to set it here rather than in the authentication function.
    // apart from the usual scenario (when an owner operates on behalf of its account),
    // the operatorAuthenticated should be cleared when about to execute the permit self-call, when
    // target contract is equal to the msg.sender in call() and batch(), or when the controlCollateral is
in
    // progress (in which case the operatorAuthenticated is not relevant)
    if (
        haveCommonOwnerInternal(onBehalfOfAccount, msgSender) || targetContract == msg.sender
            || targetContract == address(this) || contextCache.isControlCollateralInProgress()
    ) {
        executionContext =
contextCache.setOnBehalfOfAccount(onBehalfOfAccount).clearOperatorAuthenticated();
    } else {
        executionContext = contextCache.setOnBehalfOfAccount(onBehalfOfAccount).setOperatorAuthenticated();
    }

    emit CallWithContext(
        msgSender, getAddressPrefixInternal(onBehalfOfAccount), onBehalfOfAccount, targetContract,
bytes4(data)
    );

    (success, result) = targetContract.call{value: value}(data);

    executionContext = contextCache;
}
```

The impact is that a single subAccount cannot be used for another loan from a different borrow vault. The current version of the code doesn't allow that, but if that happens in the future, this issue won't allow it to do so.

Recommendation: Use evc.call and invoke disableController without any arguments.

Resolution: Fixed

5.4. Low severity

5.4.1. same subAccountId will be used for multiple loans

Severity: Low Risk

Description: The current logic of deriving a valid `subAccount` address makes it possible single subAccount to be reused across multiple loans of a single user.

```
function createIndividualLoan(LoanCreationParams calldata params) external nonReentrant returns (uint256
loanId) {//OK
    ...MORE CODE
        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);
    while (subAccount == address(0) || uint160(subAccount) <= 20) {
        subAccountId = (subAccountId + 1) % 256; // Wrap around if needed
        subAccount = LibCredifiERC20Adaptor.calculateSubAccount(params.user, subAccountId);
    }

    // Increment loan count for next sub-account
    userLoanCount[params.user]++;
    ...MORE CODE
```

Albeit highly unlikely to have an address that doesn't satisfy the while loop, we can have the following:

```
Loan 1: tries subAccountId 100, 101, 102 (invalid) → uses 103 (valid) → but sets userLoanCount = 1
Loan 2: starts with subAccountId = 1 + 100 = 101 → tries 101, 102 (invalid) → uses 103 again
Loan 3: starts with subAccountId = 2 + 100 = 102 → tries 102 (invalid) → uses 103 again
```

The root cause of this problem is that `userLoanCount[params.user]` is incremented only **once,** without taking into consideration how many loops the while has done.

Recommendation: Consider having a tracker, which is the `subAccountId`, that gives the valid address and start from the next id the next time.

Resolution: Fixed

5.4.2. `CrediFi1155::setCredifiERC20Adaptor` cannot be used when there are active loans

Severity: Low Risk

Description: `setCredifiERC20Adaptor` cannot be used when there are open loans. Otherwise, this will cause all the loans for the previous adaptor to be non-withdrawable, since the old adaptor will be the operator, restricting everyone else from performing operations with the loan.

Recommendation: Make sure to clean all the outstanding loans when you need to update the adaptor.

Resolution: Fixed

5.4.3. NFT is not burned at the end of payoffAndCloseLoan

Severity: Low Risk

Description: When the loan is closed, a comment in `payoffAndCloseLoan()` mentions that the NFT should be burned, but `CredifiERC20Adaptor` doesn't do so because it doesn't have permissions to do it.

```
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 <--------
}
```

The NFT is simply a Credifi representation, so even if it's not burned, it has a minor impact as it only inflates the total supply of the Credifi1155 token.

Recommendation: Add a public `burn` function in `Credifi1155` and call it at the end of `adaptor.closeIndividualLoan` if the loan is fully repaid.

Resolution: Fixed

5.4.4. When creating a loan in `LibAdaptor::setupSecureSubAccount` isn't needed as it does a repetitive operations

Severity: Low Risk

Description: `LibCredifiERC20Adaptor::setupSecureSubAccount`, invoked in `executeIndividualLoanSetup` , is excessive as the exact same operations - checking operator's authorization, enabling collateral and controller are already done in the `CredifiERC20Adaptor::_executeIndividualLoanSetup`.

Recommendation: Do not call `setupSecureSubAccount` from `LibCredifiERC20Adaptor::executeIndividualLoanSetup`.

Resolution: Fixed

5.4.5. Notes

Severity: Low Risk

Description:

1. `INTEREST_RATE_PER_SECOND` should be 1268391679350583460, not 1268391679350583552.

2. `updateMetadata` will make the position to differ from the actual one in Euler.

3. Loan owner can remove the Operator permissions of the Adaptor and close the loan by himself, getting the Credit tokens and leaving the storage of Adaptor unsynced with Euler.

4. important: Use the interfaces from the dedicated repos, avoid copy/pasting them manually as it can lead to mistakes, such as the one in M-02

Resolution: Fixed - 1,2,3