our shielding · Your smart contracts, our shielding · Your smart

shieldify

Credifi

SECURITY REVIEW

Date: 3 July 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Credifi

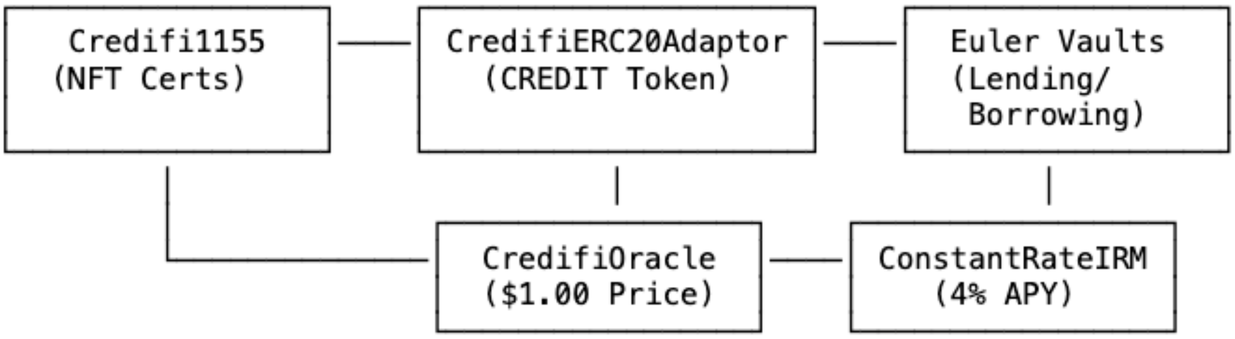
Credifi is a decentralized credit underwriting and default management protocol that enables borrowing against credit certificates represented as ERC-1155 NFTs. The protocol integrates with Euler v2 vaults to provide seamless lending with 100% collateral factors for credit-backed loans.

Overview

The Credifi protocol consists of four main components:

1. **Credifi1155**: An upgradeable ERC-1155 contract for credit certificate NFTs
2. **CredifiERC20Adaptor**: An ERC-20 wrapper that converts NFTs to fungible CREDIT tokens for use in DeFi protocols
3. **CredifiOracle**: A price oracle that provides $1.00 pricing for CREDIT tokens and USDC
4. **ConstantRateIRM**: A constant interest rate model providing 4% APY

Architecture



Credifi1155 (NFT Certs)
CredifiERC20Adaptor (CREDIT Token)
Euler Vaults (Lending/ Borrowing)
CredifiOracle ($1.00 Price)
ConstantRateIRM (4% APY)

Figure 1: Credifi's Architecture

4. Risk Classification

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

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable, and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 6 days, with a total of 96 hours dedicated by 2 researchers from the Shieldify team.

Overall, the code is well-written. The audit report flagged one Medium and four Low-severity issues, related to unchecked transfer to address, the reimplementation of an existing operator authorization logic, incorrectly overridden original owner during minting and incomplete cleaning of loans-related storage variables.

The Credifi team has done a great job with their test suite and provided exceptional support to the Shieldify researchers.

5.1 Protocol Summary

Project Name	**Credifi**
Repository	credifi
Type of Project	DeFi, ERC-1155 NFTs, Euler V2 Vaults (Lending/Borrowing)
Audit Timeline	6 days
Review Commit Hash	ba976bad4afaf2dc068ca9dcd78b38052d3686e3
Fixes Review Commit Hash	b517eff5f398dd1005af9cb356cada5ab10ab490

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
src/CredifiERC20Adaptor.sol	393
src/ConstantRateIRM.sol	43
src/Credifi1155.sol	256
src/CredifiOracle.sol	69
Total	**655**

6. Findings Summary

The following issues have been identified, sorted by their severity:

- **Medium** issues: **1**
- **Low** issues: **4**
- **Info** issues: **4**

ID	Title	Severity	Status
[M-01]	Unchecked ERC20 Transfer in Loan Repayment	Medium	Fixed
[L-01]	Custom Operator Authorization Implementation in `CredifiERC20Adaptor`	Low	Fixed
[L-02]	The Original Owner During Mint Depositing Is Incorrectly Overrid-den	Low	Fixed
[L-03]	Incomplete Storage Cleanup in `_removeTokenFromUser()`	Low	Fixed
[L-04]	Lack of Two-Step Ownership Transfer in `Credifi1155`	Low	Fixed
[I-01]	Duplicate `add()`, `addMultiple()` and Remove Whitelisting	Info	Fixed
[I-02]	Inconsistent Variable Naming in `CredifiERC20Adaptor`	Info	Fixed
[I-03]	Instances of Redundant Code	Info	Fixed
[I-04]	Inconsistent Vault Call Method in Loan Repayment	Info	Fixed

7. Findings

[M-01] Unchecked ERC20 Transfer in Loan Repayment

Severity

Medium Risk

Description

The contract performs an ERC20 token `transferFrom()` without verifying its success. While most major tokens revert on failure, some non-ERC20 standard tokens technically allow them to return false instead of reverting.

This introduces a risk where Non-Compliant Tokens may fail silently, while the protocol assumes success, potentially leading to missing or undelivered funds.

Location of Affected Code

File: src/CredifiERC20Adaptor.sol#L620

```
function _performLoanRepayment( address user, IndividualLoan storage loan
    , uint256 repayAmount, address payer, bool requireFullRepayment )
    internal returns (uint256 actualRepayAmount, uint256 remainingDebt,
    bool fullyRepaid) {
// code

// Transfer repay tokens from payer to this contract
    borrowToken.transferFrom(payer, address(this), actualRepayAmount);
// code
}
```

Recommendation

Consider using `safeTransferFrom()` instead of `transferFrom()`:

```
// Use OpenZeppelin's SafeERC20
using SafeERC20 for IERC20;

// Replace transfer with:
borrowToken.safeTransferFrom(payer, address(this), actualRepayAmount);
```

Team Response

Fixed.

[L-01] Custom Operator Authorization Implementation in `CredifiERC20Adaptor`

Severity

Low Risk

Description

The `isAuthorizedOperator()` function in `CredifiERC20Adaptor` implements a custom check for operator authorization, duplicating logic that already exists in the Ethereum Vault Connector (EVC).

Location of Affected Code

File: src/CredifiERC20Adaptor.sol#L448

```
function isAuthorizedOperator(address user) public view returns (bool) {
    bytes19 addressPrefix = evc.getAddressPrefix(user);

// Cast to extended interface to access bit field methods
    IEVCExtended evcExtended = IEVCExtended(address(evc));
    uint256 operatorBitField = evcExtended.getOperator(addressPrefix,
        address(this));

// Calculate the account ID (bit position) for this user
// Convert address prefix to owner address using the EVC pattern
    address owner = address(uint160(uint152(addressPrefix)) << 8);
    uint256 accountId = uint160(user) ^ uint160(owner);

// Check if the bit for this account is set in the operator bit field
    return (operatorBitField & (1 << accountId)) != 0;
}
```

Recommendation

Consider removing the custom logic:

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

Team Response

Fixed.

[L-02] The Original Owner During Mint Depositing Is Incorrectly Overridden

Severity

Low Risk

Description

The `Credifi1155` contract facilitates the creation of credit tokens, keeping track of each token's metadata, like amount, creation date, etc. There is also a field `originalOwner` meant to keep track of the very first holder of the token.

This field is assigned correctly during a regular `mint()`, since the overriden `_update()` sets the `originalOwner` to the receiver of the NFT if it detects a mint operation instead of a regular transfer.

However, inside `mintDepositAndBorrow()` we have:

```
TokenMetadata storage metadata = tokenMetadata[tokenId];
metadata.creditAmount = amount;
metadata.creationDate = creationDate;
metadata.certificateHash = certificateHash;
metadata.originalOwner = user;

_mint(credifiAdaptor, tokenId, 1, "");
emit TokenMinted(tokenId, certificateHash, credifiAdaptor, amount,
   creationDate);
```

The idea behind the function is to mint the token directly to the adaptor instead of minting to the user and sending it to the adaptor, but keep the user as the original owner, as that would be the case in the latter option.

However, after setting the `originalOwner` to the user's address, the call to `_mint()` would call `_update()`, which would override the original owner field with the address of the adaptor instead, leading to the metadata reporting wrong information.

Location of Affected Code

File: src/Credifi1155.sol#L509

```
function mintDepositAndBorrow( address user, uint256 amount, bytes32
   certificateHash, address collateralVault, address borrowVault, uint256
    borrowAmount ) external onlyOwner returns (uint256) {
// code
   TokenMetadata storage metadata = tokenMetadata[tokenId];
   metadata.creditAmount = amount;
   metadata.creationDate = creationDate;
   metadata.certificateHash = certificateHash;
   metadata.originalOwner = user;

   _mint(credifiAdaptor, tokenId, 1, "");
   emit TokenMinted(tokenId, certificateHash, credifiAdaptor, amount,
      creationDate);
}
```

Impact

- Wrong metadata reporting
- Off-chain monitoring would receive incorrect/unintended data

Recommendation

Either remove the owner assignment from `_update()` since it's already handled manually during minting, or add a check to skip it if the owner is already set.

Team Response

Fixed.

[L-03] Incomplete Storage Cleanup in `_removeTokenFromUser()`

Severity

Low Risk

Description

When loans are closed by calling `closeIndividualLoan()`, the system fails to fully clear all associated data. While it removes the token, related records, it neglects to delete the core loan details stored in the `userLoans[user][loanId]` mapping. This permanently occupies storage space.

The `_removeTokenFromUser()` function fails to fully clear all loan-related storage, specifically neglecting to delete the loan data from `userLoans[user][loanId]`.

Location of Affected Code

File: src/CredifiERC20Adaptor.sol#L298

```
function _removeTokenFromUser(address user, uint256 tokenId) internal {
    _removeFromArray(userDepositedTokens[user], userTokenIndex[user],
       tokenId);

    delete tokenDepositor[tokenId];
    delete mintedCreditForToken[tokenId];
    delete tokenToLoanId[tokenId];
}
```

Recommendation

Consider deleting all of the loan-related storage variables, specifically the `userLoans` mapping:

```
function _removeTokenFromUser(address user, uint256 tokenId) internal {
    _removeFromArray(userDepositedTokens[user], userTokenIndex[user],
       tokenId);

    delete tokenDepositor[tokenId];
    delete mintedCreditForToken[tokenId];
    delete tokenToLoanId[tokenId];
+   delete userLoans[user][loanId];
}
```

Team Response

Fixed.

[L-04] Lack of Two-Step Ownership Transfer in `Credifi1155`

Severity

Low Risk

Description

The `Credifi1155` contract inherits from `OwnableUpgradeable`, which allows the current owner to transfer ownership directly to a new address. However, `OwnableUpgradeable` does not include a confirmation step from the new owner.

To prevent such risks, it's recommended to use OpenZeppelin's `Ownable2Step` which introduces a two-step ownership transfer process.

Location of Affected Code

File: src/Credifi1155.sol

Recommendation

Consider using `Ownable2Step`.

```
contract Credifi1155 is Ownable2Step
```

Team Response

Fixed.

[I-01] Duplicate `add()`, `addMultiple()` and Remove Whitelisting

Severity

Informational Risk

Description

In `addToWhitelist()`, `addMultipleToWhitelist()`, and `removeFromWhitelist()`, there is no check if the address is already whitelisted. This causes unnecessary gas consumption and event emissions.

Location of Affected Code

File: src/Credifi1155.sol

```
function addToWhitelist(address account) public onlyOwner validAddress(
    account, "INVALID_ADDR") {
    whitelistedAddresses[account] = true;
    emit AddressWhitelisted(account);
}

function addMultipleToWhitelist(address[] memory accounts) public
    onlyOwner {

function removeFromWhitelist(address account) public onlyOwner {
```

Recommendation

Consider adding the existence validation check in the whitelisting related functions:

- `addToWhitelist()`

```
function addToWhitelist(address account) public onlyOwner
   validAddress(account, "INVALID_ADDR") {
 if (!whitelistedAddresses[account]) {
    whitelistedAddresses[account] = true;
    emit AddressWhitelisted(account);
 }
}
```

- `addMultipleToWhitelist()`

```
function addMultipleToWhitelist(address[] memory accounts) public
   onlyOwner {
 for (uint256 i = 0; i < accounts.length; i++) {
    address account = accounts[i];
    require(account != address(0), "INVALID_ADDR");
    if (!whitelistedAddresses[account]) {
       whitelistedAddresses[account] = true;
       emit AddressWhitelisted(account);
    }
 }
}
```

- `removeFromWhitelist()`

```
function removeFromWhitelist(address account) public onlyOwner {
  if (whitelistedAddresses[account]) {
      whitelistedAddresses[account] = false;
      emit AddressRemovedFromWhitelist(account);
  }
}
```

Team Response

Fixed.

[I-02] Inconsistent Variable Naming in `CredifiERC20Adaptor`

Severity

Informational Risk

Description

In `repayIndividualLoan()` variable `eulerVault` is too generalized and misleading because it actually uses `loan.borrowVault`. So, use the `borrowVault` declaration instead of `eulerVault`, similar to how it is declared in the `_performLoanRepayment()` function.

Location of Affected Code

File: src/CredifiERC20Adaptor.sol#L563

```
function repayIndividualLoan(address user, uint256 loanId, uint256
    repayAmount, address payer) external nonReentrant validUser(user) {
// code
    IEulerVault eulerVault = IEulerVault(loan.borrowVault);
    uint256 currentDebt = eulerVault.debtOf(loan.subAccount);
// code
}
```

Recommendation

Consider adjusting the variable names:

```
function repayIndividualLoan(address user, uint256 loanId, uint256
    repayAmount, address payer) external nonReentrant validUser(user) {
// code
-   IEulerVault eulerVault = IEulerVault(loan.borrowVault);
-   uint256 currentDebt = eulerVault.debtOf(loan.subAccount);
+   IEulerVault borrowVault = IEulerVault(loan.borrowVault);
+   uint256 currentDebt = borrowVault.debtOf(loan.subAccount);
// code
}
```

Team Response

Fixed.

[I-03] Instances of Redundant Code

Severity

Informational Risk

Description

Both the `Credifi1155` and the `CredifiERC20Adaptor` have several instances of redundant code patterns that serve no purpose, as their goal is already accomplished somewhere else in the code: - `CredifiERC20Adaptor`: whitelist checks in both `transfer()` and `transferFrom()` in the adaptor, when the `_update()` function already does all necessary whitelist checks - `Credifi1155`:

manually setting the original owner in `mint()` when `_update()` already does it if it detects a minting operation - Redundant validation checks in `depositTokenOnBehalfOf()` - the `_mint()` function already reverts if the token doesn't exist there for `Credifi1155` 's `mintDepositAndBorrow()` , the function makes sure that the `tokenID` always exists, so there's no need to recheck. - Both `validAddress` and `validUser` are performing the same check, so there's no need to declare two separate modifiers. This logic should be written once and reused.

Location of Affected Code

File: src/Credifi1155.sol

File: src/CredifiERC20Adaptor.sol

```
function mint(address to, uint256 amount, bytes32 certificateHash, bytes
   memory data) public onlyOwner returns (uint256) {
   require(to != address(0), "INVALID_ADDR");

   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; //@audit redundant, it is done inside of
      _mint > _update
// _buffer_0 gets default value (all zeros) automatically

  _mint(to, tokenId, 1, data);

  emit TokenMinted(tokenId, certificateHash, to, amount, creationDate);

  return tokenId;
}
// code

function transfer(address to, uint256 value) public virtual override
   returns (bool) {
// Allow transfers from whitelisted vaults (e.g., for vault redemptions)
   if (whitelistedVaults[msg.sender]) { //@audit redundant
      return super.transfer(to, value);
   }
   revert("Direct transfers are not allowed");
}
```

```
function transferFrom(address from, address to, uint256 value) public
    virtual override returns (bool) {
// Allow transfers to whitelisted vaults
    if (whitelistedVaults[to]) { //@audit redundant
        return super.transferFrom(from, to, value);
    }
    revert("Direct transfers are not allowed");
}
// code

function depositTokenOnBehalfOf(uint256 tokenId, address user) external
    nonReentrant validUser(user) {
   require(msg.sender == address(credifi1155), "ONLY_CREDIFI"); // Only
       Credifi1155 contract can call this function
   require(credifi1155.exists(tokenId), "Token does not exist");
   require(credifi1155.balanceOf(address(this), tokenId) == 1, "Contract
       doesn't own this token");
   require(tokenDepositor[tokenId] == address(0), "Token already deposited
       ");
// code

modifier validAddress(address addr, string memory errorMsg) {
    require(addr != address(0), errorMsg);
    _;
}

modifier validUser(address user) {
    require(user != address(0), "Invalid user address");
    _;
}
```

Recommendation

Consider removing the redundant instances.

Team Response

Fixed.

[I-04] Inconsistent Vault Call Method in Loan Repayment

Severity

Informational Risk

Description

The contract consistently uses the `_secureVaultCall()` pattern for all interactions with the vault via `evc.call()`. However, in the `_performLoanRepayment()` function, it directly calls `evc.call()` without using `_secureVaultCall()`, leading to an inconsistency in the codebase.

Location of Affected Code

File: src/CredifiERC20Adaptor.sol#L628

```
function _performLoanRepayment(
    address user,
    IndividualLoan storage loan,
    uint256 repayAmount,
    address payer,
    bool requireFullRepayment
) internal returns (uint256 actualRepayAmount, uint256 remainingDebt,
    bool fullyRepaid) {
// code
    bytes memory repayCalldata =
        abi.encodeWithSignature("repay(uint256,address)", actualRepayAmount
            , loan.subAccount);
        evc.call(loan.borrowVault, address(this), 0, repayCalldata);
// code
}
```

Recommendation

Consider using `_secureVaultCall()`:

```
bytes memory repayCalldata =
     abi.encodeWithSignature("repay(uint256,address)", actualRepayAmount,
        loan.subAccount);
+   _secureVaultCall(loan.borrowVault, address(this),0,repayCalldata);
```

Team Response

Fixed.

, our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Thank you!