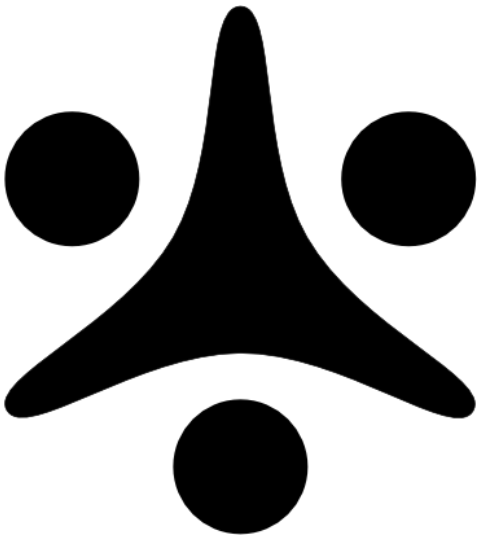

SmartCredit.io Security Analysis

by Pessimistic

This report is public.

Published: April 23, 2021

Abstract

In this report, we consider the security of smart contracts of SmartCredit.io project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, security audit is not an investment advice.

Summary

In this report, we considered the security of SmartCredit smart contracts. We performed our audit according to the procedure described below.

The initial audit showed one critical issue that could lead to a payment failure. Also, many issues of medium and low severity were found. The project had insufficient documentation, the tests coverage was low, some tests did not pass.

After the initial audit, the code base was updated to the latest version. The payment failure issue and most of ither issues were fixed, some of the issues were commented. The overall code quality and architecture of the project were improved, additional documentation was provided, and the tests coverage was improved.

However, new ERC20 standard violation issue appeared in the code with the update.

The owner of the contract has special powers: after the loan request has been matched, the owner of the contract has full control over users' collaterals.

General recommendations

We recommend fixing the rest of the issues. We also recommend adding continuous integration to the project: running tests, code coverage, and security tools like Slither.

Project overview

Project description

For the audit, we were provided with SmartCredit project on a private BitBucket repository, commit 7bbfa40d56877c8b656488fc52baeb2d335c3335.

The documentation for the project was provided as **SmartCredit.io-User-Manual.pdf** file, sha1sum is 1f1c369b4bf7a59453339922246c53dac1ad7e8d.

The tests coverage is low, some tests fail.

The total LOC of audited sources is 2851.

Latest version of the code

For the recheck, we were provided with the updated code base, commit 74e4c35f2a29b18ce5bc6b35dc0ab21fe4099f4a.

Also, new documentation was provided: Activity Diagrams.pdf, sha1sum is 00b2fbc266bb3ec3cecaff7f62311f9740ad4893.

All the failing tests were fixed. The coverage was increased up to 79.9%.

Procedure

In our audit, we consider the following crucial features of the code:

1. Whether the code is secure.
2. Whether the code corresponds to the documentation (including whitepaper).
3. Whether the code meets best practices.

We perform our audit according to the following procedure:

- Automated analysis
  - We scan project's code base with automated tools: Slither and SmartCheck.
  - We manually verify (reject or confirm) all the issues found by tools.
- Manual audit
  - We manually analyze code base for security vulnerabilities.
  - We assess overall project structure and quality.
- Report
  - We reflect all the gathered information in the report.

Manual analysis

The contracts were completely manually analyzed, their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger smart contracts security. We highly recommend fixing them.

Payment failure (fixed)

`payToLenders()` function of **ETHLoan** contract sends Ether inside a loop. If any of recipients fails to accept the payment, the whole transaction fails. As a result, lenders will be unable to receive their Ether.

We recommend implementing pull pattern.

> *Comment from developers: Credit coin implementation is removed and transfer to lenders happen when user repays a loan.*

The issue has been fixed and is not present in the latest version of the code.

Medium severity issues

Medium issues can influence project operation in current implementation. We highly recommend addressing them.

Overpowered roles

An admin role has full control over users' collaterals.

This can result in undesirable consequences for the project and its users if an admin's private keys become compromised.

We recommend designing contracts in a trustless manner or implementing proper key management, e.g. multisig.

> *Comment form developers: Platform owner cannot take collateral from **ERC20Collateral**, **ERC20Fund**, and **ETHFund**. The **LoanContract** will be created after the loan has been matched, so the platform can access the assets for the liquidation.*

Bad design (fixed)

CreditCoin contract has significand design flaws and logic errors.

- In `_transfer()` function, if `MAX_HOLDERS` owners of a loan is reached and the recipient is one of them, the transfer is forbidden. However, the case seems legit, and it does not increase the number of the owners of the loan.
- `_transfer()` function can fail with assert-style error. Consider this scenario: user A has two loans with 1 token each and tries to transfer these 2 tokens to user B. The code enters `while` loop at line 233, `i == 0`. If the first loan has too many owners, the `while` loop goes to the next iteration with `i == 1`. The second loan competes the transfer successfully, therefore `i` value does not change (line 265). After `removeCreditLoan()` call, `creditedLoan[A][1] == 0`. Thus, the code gets into `else` condition with `i == 2`, line 282. In the next iteration, this causes `out of boundary` error at line 234 and therefore fails with assert-style error.
- The design that includes the use of `MAX_HOLDERS` constant is dubious. It is error-prone, hard to test, and affects user experience. We recommend removing and redesigning `DAILOoan.payToLenders()` so that it does not run out of gas.

***CreditCoin** contract was removed from the code base. Thus, the issues are not present in the latest version of the code.*

ERC20 standard violation

- EIP-20 states:

```
Callers MUST handle false from returns (bool success). Callers MUST
NOT assume that false is never returned!
```

  However, in the code of **DAILoan**, **USDCLoan**, **USDTLoan**, and **ETHLoan** contracts, the returned values of `ERC20.transfer()` and `ERC20.transferFrom()` functions are not checked in `validateRepayment()`, `increaseCollateral()`, `transfer()`, and `releaseCollateral()` functions.

  The returned values are not checked for functions `ERC20Interface.approve()`, `ERC20Interface.transfer()`, and `ERC20Interface.transferFrom()` in **DAIFIF**, **USDCFIF**, **USDTFIF**, and **ETHFIF** contracts.

  The returned value of `ERC20.approve()` is not checked in **StableCoinLiquidate** contract.

  The issues have been fixed and are not present in the latest version of the code.

- According to EIP-20:

```
Transfers of 0 values MUST be treated as normal transfers and fire
the Transfer event.
```

  However, `_transfer()` function of **CreditCoin** contract does not allow zero value transfers at line 224:

```
require(amount > 0, "Credit Coin: Zero Value transfers not allowed");
```

  CreditCoin *contract was removed from the code base. Thus, the issue is not present in the latest version of the code.*

- In **ERC20Liquidate** contract at line 46, the returned value of `transfer()` call is not checked inside `withdraw()` function. The issue appeared in the code base after the initial audit.

> *Comment form developers: `transfer()` will be replaced with `safeTransfer()`. However, this function is callable only by the admin, and it is called only for withdrawing the liquidation fees. It is not a function what a regular user can call; And even admin will call it very rarely.*

We highly recommend following ERC20 standard to minimize integration issues.

Returned value not checked (fixed)

Compound functions do not revert in case of error but return error code. Therefore, the returned values of such calls must be checked. However, in `transferToCompound()` function of **DAIFIF**, **USDCFIF**, **USDTFIF**, and **ETHFIF** contracts, the returned value is not checked.

The issue has been fixed and is not present in the latest version of the code.

Improper variable type

In **SmartCreditRegistry** contract, `_keys` array is iterated in `for` loop with `uint8` loop counter at lines 24, 32, 53, and 62:

```
for (uint8 i = 0; i < _keys.length; i++)
```

This limits the number of processable `_keys` values to `256` which might affect both on-chain and off-chain code. We recommend using loop counter of `uint256` type.

> *Comment form developers: We have only 5 to 8 variables here. So, we will not touch the 256 limit.*

Bugs (fixed)

- In `defaultRepayByLPF()` function of **ETHLoan** contract, the loan status is only updated if `loan.status != DEFAULTPAID`. However, in **DAILoan**, **USDCLoan**, and **USDTLoan** contracts, this check is missing.
- Function `investToCompound()` of **DAIFIF** contract should use `_bucketInvested` value instead of `_bucketInvested` at lines 108 and 110.
- If-else block has incorrect condition in `_transfer()` function of **CreditCoin** contract at line 242: it should be `amount - transferred < countSender` instead of `amount < countSender`.

The issues have been fixed and are not present in the latest version of the code.

Discrepancy with the documentation (fixed)

According to documentation, **CreditCoin** contract should have token name `Credit Coin`. However, in the code, the name of the token is not declared.

***CreditCoin** contract was removed from the code base. Thus, the issues are not present in the latest version of the code.*

Insufficient documentation (fixed)

The provided documentation is incomplete and outdated. The documentation helps auditors to verify the correctness of the code. It also helps other developers with token integration.

We strongly recommend keeping the documentation to the project up to date.

Additional documentation for the project was provided.

Tests issues (fixed)

The provided code has tests. However, the coverage is about 42% and is not measured regularly. Moreover, 18 tests fail.

Testing is crucial for code security and audit does not replace tests in any way.

We highly recommend checking and improving test coverage regularly as the project grows.

The issues have been fixed and are not present in the latest version of the code. All tests pass, the coverage is 79.9%.

Low severity issues

Low severity issues can influence project operation in future versions of code. We recommend taking them into account.

Code quality

- There are three groups of contracts that share large portions of the code within each group. This results in issues duplications and significantly affects the maintainability of the code. We recommend unifying and reusing the contracts to minimize code duplication. These groups are:
    - **StableCoinFund** and **ERC20Collateral** contracts are almost identical and share a large portion of the code with **ETHFund** contract.
    - **DAILoan**, **USDCLoan**, **USDTLoan**, and **ETHLoan** contracts.
    - **DAIFIF**, **USDCFIF**, **USDTFIF**, and **ETHFIF** contracts.

  Thus, in most cases, the issues described for each of these contracts apply to the whole group that the contract belongs to.

  The issues have been fixed and are not present in the latest version of the code.
- Since Solidity 0.6.5, the keyword `immutable` was introduced for variables and constants. We recommend upgrading the project to compile with Solidity version 0.6.5 or higher, and declaring constants or variables as `immutable` to improve code readability and optimize gas consumption:
    - In **ERC20Collateral** contract at line 21.
    - **DAILoan**, **USDCLoan**, and **USDTLoan** contracts differ from each other with values of the constants. Consider declaring these constants as `immutable` and declaring them in the `constructor`.

  The issues have been fixed and are not present in the latest version of the code.
- There is a misleading comment in ERC20Collateral contract at lines 50–51. The comment states: `using token symbol as identifier in mapping`. However, this is not implemented in the code.

  The issue has been fixed and is not present in the latest version of the code.
- In **ETHFund** contract, `_amount` parameter of `deposit()` function is redundant.

  The issue has been fixed and is not present in the latest version of the code.
- Consider combining `INACTIVE`, `ACTIVE`, `REPAID`, `DEFAULT`, and `DEFAULTPAID` constants from **DAILoan**, **ETHLoan**, **USDCLoan**, and **USDTLoan** contracts and `LIQUIDATION_NOT_REQUIRED`, `LIQUIDATION_STARTED`, and `LIQUIDATION_COMPLETE` constants from **Liquidation** contract into separate `enum`s.

  The issue has been fixed and is not present in the latest version of the code.

- `validateRepayment()` modifier in **DAILoan**, **USDCLoan**, and **USDTLoan** contracts complicates the code and leads to CEI pattern violation as it has side effects.

  The issue has been fixed and is not present in the latest version of the code.

- `defaultRepayByLPF()` function in **DAILoan**, **ETHLoan**, **USDCLoan**, and **USDTLoan** contracts should not be `payable`.

  The issue has been fixed and is not present in the latest version of the code.

- `receive()` function of **ETHFIF** contract should not be `virtual`.

  The issue has been fixed and is not present in the latest version of the code.

- avoid specifying `gas` in calls like in `transferToCompound()` function of **ETHFIF** contract.

  The issue has been fixed and is not present in the latest version of the code.

- In **CreditCoin** contract, consider using `assert()` instead of `revert()` at line 292 if the condition is supposed to be unreachable.

  ***CreditCoin** contract was removed from the code base. Thus, the issues are not present in the latest version of the code.*

- In `removeCreditLoan()` and `removeCreditOwner()` functions of **CreditCoin** contract, consider using `array.pop()` instead of `delete array[last]`.

  ***CreditCoin** contract was removed from the code base. Thus, the issues are not present in the latest version of the code.*

- In `transferLiquidatedAmount()` function of **StableCoinLiquidate** contract, consider removing the condition at lines 26–28 as ERC20 tokens should never revert here.

  The issue has been fixed and is not present in the latest version of the code.

- `withdraw()` function of **SmartCredit** contract should not be payable.

  The issue has been fixed and is not present in the latest version of the code.

- According to Naming Conventions of Solidity Style Guide, `Contract and library names should also match their filenames`. However, the convention is not respected for the following files:
  - Interfaces/ICErc20.sol
  - Interfaces/ICEther.sol
  - Escrow/Fund/ETHBased.sol
  - Escrow/Fund/StableCoinBased.sol
  - Escrow/Liquidation/ETHBased.sol
  - Escrow/Liquidation/StableCoinBased.sol

  The issues have been fixed and are not present in the latest version of the code.

- The code has many `todo`s all over the project.

  The issue has been fixed and is not present in the latest version of the code.

- In the project, developers often use several constants instead of a single `enum`.

  The issue has been fixed and is not present in the latest version of the code.

- Consider declaring functions as `external` instead of `public` where possible.

  The issue has been fixed and is not present in the latest version of the code.

- In the constructor of **SmartCredit** contract, four roles granted to the same account.

  Consider distributing these roles among different accounts.

  > *Comment from developers: Roles can be granted and revoked using **AccessControl** contract.*

- Avoid using low-level calls like in `investFixedIncomeFundToCompound()` function of **SmartCredit** contract when the function can be called explicitly. Using low-level calls significantly affects the readability of the code.

  > *Comment form developers: will be fixed in the next release.*

- consider replacing constants `BUCKET1`, `BUCKET2`, `BUCKET3`, `BUCKET4`, `RANGE1`, `RANGE2`, `RANGE3`, and `RANGE4` in **DAIFIF**, **ETHFIF**, **USDCFIF**, and **USDTFIF** contracts with an `enum` and a `library` combination via `using for` to make contract more granular.

- Using `uint8` for loop counter does not save gas but can result in serious issues.

This analysis was performed by Pessimistic:

Evgeny Marchenko, Senior Security Engineer
Vladimir Tarasov, Security Engineer
Boris Nikashin, Analyst
Alexander Seleznev, Founder

April 23, 2021