HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Lucidao
Date: 17 October, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for Lucidao
Approved By	Grzegorz Trawiński | Solidity SC Lead Auditor at Hacken OÜ
Tags	ERC20; ERC721; Lending
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://lucidao.com
Changelog	04.10.2023 - Initial Review 17.10.2023 - Second Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by Lucidao (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Lucidao is a lending protocol with the following contracts:

- *Lending* – a contract that allows users to request loans by depositing NFTs as collateral and other users to fulfill the request by depositing the desired tokens. If the loan is not paid back by the borrower before the deadline the lender can either withdraw the NFTs or any other user is able to liquidate the loan, by depositing the tokens back to the lender and leaving with the NFTs.

Privileged roles

- The owner of the *Lending* contract can arbitrarily modify the price index address, governance treasury address, repay grace period, repay grace fee, protocol fee, liquidation fee, base origination fee, whitelisted tokens, loan types/interest rates, fee reduction factor and origination fee ranges.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **7** out of **10**.

- Functional requirements are partially provided.
  - Business logic is not provided.
  - Use cases are not provided.
- Technical description is provided.

Code quality

The total Code Quality score is **10** out of **10**.

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- All protocol features and cases are covered with tests.

Security score

As a result of the audit, the code contains **1** medium severity issue. The security score is **9** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.5**. The system users should acknowledge all the risks summed up in the risks section of the report.

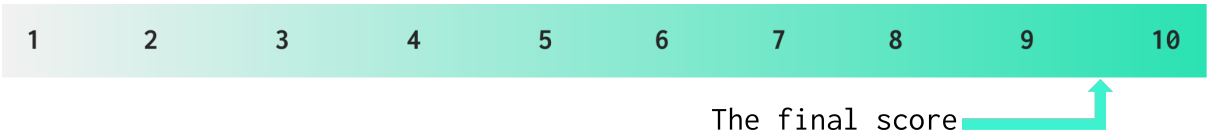


Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
4 October 2023	6	6	1	0
17 October 2023	0	1	0	0

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Risks

- The correct valuation of the assets depends on the data provided by external sources (e.g the PriceIndex contract). The owner is able to arbitrarily change the trusted price index address.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed	
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Not Relevant	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	
Unchecked Call Return Value	The return value of a message call should be checked.	Not Relevant	
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed	
SELFDESTRUCT Instruction	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Not Relevant	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Passed	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed	
Signature Unique Id	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Passed	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Flashloan Attack	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used. Contracts shouldn't rely on values that can be changed in the same transaction.	Passed	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Not Relevant	
Gas Limit and Loops	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed	
Style Guide Violation	Style guides and best practices should be followed.	Passed	
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed	
Secure Oracles Usage	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Passed	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Findings

Critical

No critical severity issues were found.

High

H01. LiquidationFee and GraceFee Miss Percentage Divisor

Impact	High
Likelihood	Medium

The Lending solution applies multiple various fees during loan repayment. The *GraceFee* is applied whenever the borrower attempts to pay off the loan during the *grace* period. Alternatively, the *LiquidationFee* is applied whenever the liquidator attempts to liquidate the overdue loan. However, both of these fees are missing the percentage divisor in the calculation formula. As a consequence, in both cases the payer must pay an overestimated value of tokens.

GraceFee calculation within *repayLoan()* function.

```
if (block.timestamp > loan.startTime + loan.duration) {
    platformFee += (totalPayable * repayGraceFee) / PRECISION;
}
```

```
   function getLiquidationFee(uint256 _borrowedAmount) public view returns
(uint256) {
       return (_borrowedAmount * liquidationFee) / PRECISION;
   }
```

In contrast, the *OriginationFee* and *ProtocolFee* are divided by 100 in their calculations.

Based on the unit tests from the *Lending.t.sol* file, the *GraceFee* should be 2.5% whereas the *LiquidationFee* should be 5%, but actual applied fees are 250% and 500% respectively.

Proof of Concept:

Instance 1 - grace fee:

1. As a deployer, deploy the solution. Set the USDT token as a lending token. Set any ERC721 as NFT collateral. Set the grace period for 5 days. Set the grace fee to 25000, which represents 2.5%.

2. As a borrower *requestLoan* for 100e6 USDT tokens with any NFT as collateral.
3. As a lender *acceptLoan*.

```
ERC20(loan.token).balanceOf(borrower)
1100000000
ERC20(loan.token).balanceOf(lender)
900000000
ERC20(loan.token).balanceOf(governanceTreasury)
0
```

4. Forward blockchain time for the period of loan duration and 4 days, so the time is within the grace period.
5. As a borrower *repayLoan*. Observe the USDT's balances. Note that the treasury received platform fee and overestimated grace fee, which is equal to around 299e6 of USDT tokens.

```
ERC20(loan.token).balanceOf(borrower)
684164650
ERC20(loan.token).balanceOf(lender)
1016050000
ERC20(loan.token).balanceOf(governanceTreasury)
299785350
```

Instance 2 - liquidation fee:

1. As a deployer, deploy the solution. Set the USDT token as a lending token. Set any ERC721 as NFT collateral. Set the grace period for 5 days. Set the liquidation fee to 50000, which represents 5%.
2. As a borrower *requestLoan* for 100e6 USDT tokens with any NFT as collateral.
3. As a lender *acceptLoan*.

```
ERC20(loan.token).balanceOf(borrower)
1100000000
ERC20(loan.token).balanceOf(lender)
900000000
ERC20(loan.token).balanceOf(governanceTreasury)
0
ERC20(loan.token).balanceOf(liquidator)
1000000000
```

4. Forward blockchain time for the period of loan duration and 6 days, so the time is within the liquidation period.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

5. As a liquidator *liquidateLoan*. Observe the USDT's balances. Note that the treasury received platform fee and overestimated liquidation fee, which is equal to around 502e6 of USDT tokens.

```
ERC20(loan.token).balanceOf(borrower)
1100000000
ERC20(loan.token).balanceOf(lender)
1016050000
ERC20(loan.token).balanceOf(governanceTreasury)
502760100
ERC20(loan.token).balanceOf(liquidator)
381189900
```

Path: ./src/Lending.sol: repayLoan(), getLiquidationFee()

Recommendation: It is recommended to apply percentage division in every fee applied across the solution.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: Every occurrence of the percentage division is now removed from the code. All configuration input data must now be provided with normalized values with the reference *PRECISION* constant set to 10000. E.g. 4% fee is now represented as a value of 400.

Medium

M01. GraceFee Is Applied on Top of Other Fees

Impact	Low
Likelihood	High

The Lending solution applies multiple various fees during loan repayment. The *GraceFee* is applied whenever the borrower attempts to pay off the loan during the *grace* period. However, the *GraceFee* is applied after calculating the *totalPayable*, which represents the loan amount, accumulated interest and origination fee. Thus, it includes additional value accrued from already charged interest and origination fee.

In contrast, the *LiquidationFee* is calculated based only on the loan amount.

```
function repayLoan(uint256 _loanId) external nonReentrant {
        Loan storage loan = loans[_loanId];

        require(loan.borrower != address(0) && loan.lender != address(0), "Lending:
```

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

```
invalid loan id");
        require(!loan.paid, "Lending: loan already paid");
        require(block.timestamp < loan.startTime + loan.duration +
repayGracePeriod, "Lending: too late");

        uint256 totalPayable = loan.amount
            + getDebtWithPenalty(
                loan.amount, loan.interestRate + protocolFee, loan.duration,
block.timestamp - loan.startTime
            ) + getOriginationFee(loan.amount);
        uint256 lenderPayable = loan.amount
            + getDebtWithPenalty(loan.amount, loan.interestRate, loan.duration,
block.timestamp - loan.startTime);
        uint256 platformFee = totalPayable - lenderPayable;

        loan.paid = true;

        IERC20(loan.token).safeTransferFrom(msg.sender, loan.lender,
lenderPayable);

        if (block.timestamp > loan.startTime + loan.duration) {
            platformFee += (totalPayable * repayGraceFee) / PRECISION;
        }

        IERC20(loan.token).safeTransferFrom(msg.sender, governanceTreasury,
platformFee);

        IERC721(loan.nftCollection).safeTransferFrom(address(this), loan.borrower,
loan.nftId);

        emit LoanRepayment(_loanId, lenderPayable + platformFee, platformFee);
    }
```

Proof of Concept: n/a

Path: ./src/Lending.sol: repayLoan()

Recommendation: It is recommended to verify whether grace fee should be applied on top of all fees or only on borrowed amount.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: Grace fee is no more applied on top of protocol fee and origination fee. However, it is applied on top of borrowed amount and interest accrued.

M02. Solution Does Not Support Inclusive fee-on-transfer

Impact	High
Likelihood	Low

The Lending solution uses ERC20 tokens for borrowing and repayment. However, it was identified that it does not support inclusive fee-on-transfer tokens. Fee-on-transfer tokens apply a small fee during the transfer, which is usually consumed by the protocol provider. Thus, the receiver may receive a decreased amount than originally declared prior to transfer.

Within the *Lending* solution it may manifest itself as inaccurate accounting between the value declared in the loan and the actual value received by the borrower.

In the Ethereum ecosystem DGX and CGT are examples of fee-on-transfer tokens. On Ethereum, the USDT stablecoin also has the fee feature implemented, which is disabled at the time of writing. On the Polygon,the USDT token is deployed as upgradeable.

```
function acceptLoan(uint256 _loanId) external nonReentrant {
        Loan storage loan = loans[_loanId];

        require(loan.borrower != address(0) && loan.lender == address(0), "Lending:
invalid loan id");
        require(!loan.cancelled, "Lending: loan cancelled");
        require(loan.deadline > block.timestamp, "Lending: loan acceptance deadline
passed");

        loan.lender = msg.sender;
        loan.startTime = block.timestamp;

        IERC20(loan.token).safeTransferFrom(msg.sender, loan.borrower,
loan.amount);
        IERC721(loan.nftCollection).safeTransferFrom(loan.borrower, address(this),
loan.nftId);

        emit LoanAccepted(_loanId, loan.lender, loan.startTime);
    }
```

Proof of Concept:

1. By means of the favorite testing framework fork Ethereum blockchain.

2. As a deployer, deploy the solution. Set the USDT token as a lending token.
3. As USDT token owner setup fee-on-transfer feature.
4. As a borrower *requestLoan* for 100e6 USDT tokens with any NFT as collateral.
5. As a lender *acceptLoan*. Observe the USDT's balances. Note that the borrower received less tokens than agreed in the loan agreement.

```
ERC20(loan.token).balanceOf(borrower)
1099850000
ERC20(loan.token).balanceOf(lender)
900000000
```

Path: ./src/Lending.sol: acceptLoan(), repayLoan(), liquidateLoan()

Recommendation: It is recommended to identify the exact amount received by the receiver as a difference between the token balance before and after the transfer transaction is made.

Found in: 6460ac1

Status: **Mitigated**

Remediation: The client's team acknowledged this finding and provided some mitigation off-chain. The inclusive fee-on-transfer scenario is now documented in the client's FAQ. Also, it is being considered to add Transaction Simulator to solution's frontend to help end users better understand the outcomes before confirming transactions.

M03. Blacklisted Lender May Prevent Debt Payoff and Liquidation

Impact	**High**
Likelihood	Low

The Lending solution allows lender to lend an amount of tokens to the borrowers, which should be paid off within the time of loan duration or grace period. Based on the unit tests from the *Lending.t.sol* file, the loan duration can be set up to 18 months. Also, the solution's owner can whitelist lending tokens by means of the *setTokens()* function. The USDC stablecoin has a blacklist feature implemented for both transfer sender and receiver, preventing blacklisted users from the protocol usage. Assuming that USDC is allowed to lend in the solution, in rare cases, the blacklisted lender may prevent debt payoff and liquidation. In such case, the borrower will not be capable of paying back the loan and collecting the NFT collateral.

The USDT stablecoin has a blacklist feature implemented as well, but it is applicable only for the sender, not for the receiver.

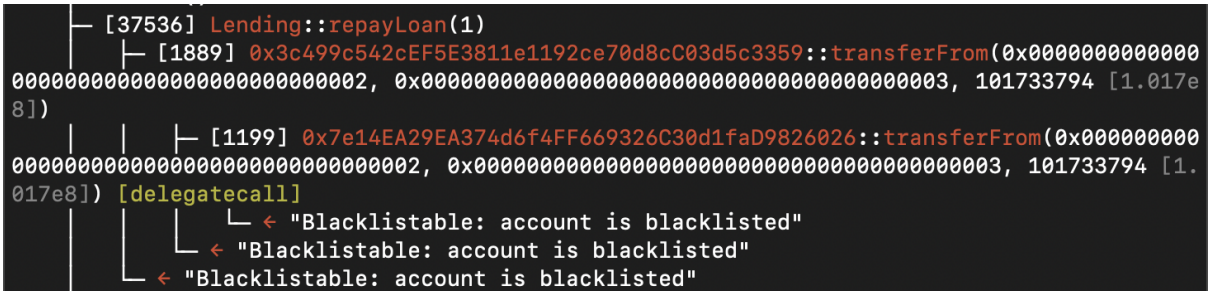
The deployment of native USDC stablecoin on Polygon is planned on 10th of October, 2023.

```
function repayLoan(uint256 _loanId) external nonReentrant {
        Loan storage loan = loans[_loanId];

        require(loan.borrower != address(0) && loan.lender != address(0), "Lending:
invalid loan id");
        require(!loan.paid, "Lending: loan already paid");
        require(block.timestamp < loan.startTime + loan.duration +
repayGracePeriod, "Lending: too late");

        uint256 totalPayable = loan.amount
            + getDebtWithPenalty(
                loan.amount, loan.interestRate + protocolFee, loan.duration,
block.timestamp - loan.startTime
            ) + getOriginationFee(loan.amount);
        uint256 lenderPayable = loan.amount
            + getDebtWithPenalty(loan.amount, loan.interestRate, loan.duration,
block.timestamp - loan.startTime);
        uint256 platformFee = totalPayable - lenderPayable;

        loan.paid = true;

        IERC20(loan.token).safeTransferFrom(msg.sender, loan.lender,
lenderPayable);

        if (block.timestamp > loan.startTime + loan.duration) {
            platformFee += (totalPayable * repayGraceFee) / PRECISION;
        }

        IERC20(loan.token).safeTransferFrom(msg.sender, governanceTreasury,
platformFee);

        IERC721(loan.nftCollection).safeTransferFrom(address(this), loan.borrower,
loan.nftId);

        emit LoanRepayment(_loanId, lenderPayable + platformFee, platformFee);
    }
```

Proof of Concept:

1. By means of the favorite testing framework fork Ethereum blockchain.
2. As a deployer, deploy the solution. Set the USDC token as a lending token.

3. As a borrower *requestLoan* for 100e6 USDC tokens with any NFT as collateral.
4. As a lender *acceptLoan*.
5. As the USDC token owner blacklists the lender account.
6. As a borrower, attempt to *repayLoan*. Observe that the transaction reverts with *Blacklistable: account is blacklisted* error message.



Path: ./src/Lending.sol: repayLoan(), liquidateLoan()

Recommendation: It is recommended to implement a failover mechanism, which will transfer lender's fund into the escrow account, in case of any issues with fund transferring, such as blacklisting of the user.

Found in: 6460ac1

Status: **Fixed** (Revised commits: 7c2e10, b1af05)

Remediation: The failover mechanism is now implemented. Whenever transfer fails, the lender's tokens are transferred to the Lender contract. Then, the lender can retrieve such tokens by means of the *withdrawStuckToken()* function.

M04. OriginationFeeRanges Does Not Support Multiple Lending Tokens

Impact	**High**
Likelihood	**Low**

The Lending solution applies multiple various fees during loan repayment including *OriginationFee*. The *OriginationFee* is scaled based on the *OriginationFeeRanges* and the *feeReductionFactor*. The algorithm compares the loan amount directly with the fixed values set in origination fee ranges to determine whether the reduction is applicable.

This limitation makes the solution to use only tokens with the same value and decimal points, otherwise the *OriginationFee* may be calculated incorrectly.

The USDT and USDC stablecoins are redeemable 1:1 for US dollars and they have set 6 decimal points. In contrast, the DAI stablecoin has

the same value, but it has set 18 decimal points. Alternatively, the wrapped ETH token has significantly higher value than mentioned above stablecoins.

```
function getOriginationFee(uint256 _amount) public view returns (uint256) {
        uint256 originationFee = baseOriginationFee;

        for (uint256 i = 0; i < originationFeeRanges.length; i++) {
            if (_amount < originationFeeRanges[i]) {
                break;
            } else {
                originationFee = (originationFee * PRECISION) / feeReductionFactor;
            }
        }
        return (_amount * originationFee) / 100 / PRECISION;
    }
```

Proof of Concept: n/a

Path: ./src/Lending.sol: repayLoan(), liquidateLoan()

Recommendation: It is recommended to normalize the value presented within *OriginationFeeRanges* parameter to the type and value of lending token, before doing the actual comparison within the *getOriginationFee()* function.

Found in: 6460ac1

Status: Fixed (Revised commit: 7c2e10)

Remediation: The *OriginationFeeRanges* represents now not normalized value, which is multiplied by the borrowed token decimals points within the *getOriginationFee()* function.

M05. Collateral Valuation Is Not Checked Within acceptLoan() Function

Impact	Medium
Likelihood	Medium

The Lending solution allows for borrowing the lending token from the lenders. To start a loan, the borrower must firstly call *requestLoan()* to set up loan terms, including the loan request's deadline. Within this function, the NFT value is checked with the *IPriceIndex* contract. Then, the lender must call *acceptLoan()* to start the loan and transfer the funds. However, in between of these two function calls, the NFT value can be decreased, making the collateral worth less than initially assumed, assuming that records in *IPriceIndex* are frequently updated.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

```
function requestLoan(
        address _token,
        uint256 _amount,
        address _nftCollection,
        uint256 _nftId,
        uint256 _duration,
        uint256 _deadline
    ) external nonReentrant {
        require(allowedTokens[_token], "Lending: borrow token not allowed");
        require(aprFromDuration[_duration] != 0, "Lending: invalid duration");
        require(_amount > 0, "Lending: borrow amount must be greater than zero");
        require(_deadline > block.timestamp, "Lending: deadline must be after
current timestamp");

        IPriceIndex.Valuation memory valuation =
priceIndex.getValuation(_nftCollection, _nftId);

        require(_amount <= (valuation.price * valuation.ltv) / 100, "Lending:
amount greater than max borrow");

        Loan storage loan = loans[++lastLoanId];
        loan.borrower = msg.sender;
        loan.token = _token;
        loan.amount = _amount;
        loan.nftCollection = _nftCollection;
        loan.nftId = _nftId;
        loan.duration = _duration;
        loan.collateralValue = valuation.price;
        loan.interestRate = aprFromDuration[_duration];
        loan.paid = false;
        loan.deadline = _deadline;
        loan.cancelled = false;

        emit LoanCreated(
            lastLoanId,
            loan.borrower,
            loan.token,
            loan.amount,
            loan.nftCollection,
            loan.nftId,
            loan.duration,
            loan.interestRate,
            loan.collateralValue,
            loan.deadline
        );
    }
```

Path: ./contracts/contract.sol : requestLoan(), acceptLoan()

Recommendation: It is recommended to perform additional verification of NFT price valuation within the *acceptLoan()* function to check, whether the loan value is still adequate.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: The NFT collateral value is now checked in both *requestLoan()* and *acceptLoan()* functions.

M06. IPriceIndex Serves Single Value Without Normalization

Impact	Medium
Likelihood	Medium

The Lending solution allows for borrowing the lending token with the collateral in the form of a valuable NFT token. The solution is designed to support multiple lending tokens, if only whitelisted by the solution's owner. Within the *requestLoan()* function, the NFT value is checked with the *IPriceIndex* contract. However, this contract neither does require any information about ERC20 token used for valuation nor it provides such information in the returned struct. Thus, it can be assumed that it returns the valuation in a single predefined off-chain cryptocurrency. This might lead to incorrect value comparison results, e.g. when an amount of whitelisted DAI (18 decimals) lending token is compared with collateral valuated in USDT (6 decimals) stablecoin. Ultimately, it may lead to the situation that the loan is under- or over-collateralized.

```
interface IPriceIndex {
   struct Valuation {
       uint256 timestamp;
       uint256 price;
       uint256 ltv;
   }

   function getValuation(address nftCollection, uint256 tokenId) external view
returns (Valuation calldata valuation);
}
```

Path: ./contracts/contract.sol : requestLoan()

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Recommendation: It is recommended to either normalize the value provided by the *IPriceIndex* contract to the lending token or redesign the *IPriceIndex* solution to return valuation in requested ERC20 token.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: The *IPriceIndex* contract now returns not normalized value, which is multiplied by the borrowed token decimals points for collateral value verification.

Low

L01. Allowed Tokens Are Checked Too Early in the Lending Process

Impact	Low
Likelihood	Low

The Lending solution allows for borrowing the lending token from the lenders. The solution's owner must whitelist lending tokens by means of the *setTokens()* function prior to usage. To start a loan, the borrower must firstly call *requestLoan()* to set up loan terms. Within this function, the lending token is checked against the *allowedTokens* collection. Then, the lender must call *acceptLoan()* to start the loan and transfer the funds. However, in between of these two function calls, the solution's owner may revoke lending token whitelisting with the *unsetTokens()* function due to any reason, including security incidents. Thus, the requested loan can still be accepted before the *deadline* with obsolete token, which might lead to uncertain issues.

```
function requestLoan(
        address _token,
        uint256 _amount,
        address _nftCollection,
        uint256 _nftId,
        uint256 _duration,
        uint256 _deadline
    ) external nonReentrant {
        require(allowedTokens[_token], "Lending: borrow token not allowed");
        require(aprFromDuration[_duration] != 0, "Lending: invalid duration");
        require(_amount > 0, "Lending: borrow amount must be greater than zero");
        require(_deadline > block.timestamp, "Lending: deadline must be after
current timestamp");

        IPriceIndex.Valuation memory valuation =
priceIndex.getValuation(_nftCollection, _nftId);
```

```
        require(_amount <= (valuation.price * valuation.ltv) / 100, "Lending:
amount greater than max borrow");

        Loan storage loan = loans[++lastLoanId];
        loan.borrower = msg.sender;
        loan.token = _token;
        loan.amount = _amount;
        loan.nftCollection = _nftCollection;
        loan.nftId = _nftId;
        loan.duration = _duration;
        loan.collateralValue = valuation.price;
        loan.interestRate = aprFromDuration[_duration];
        loan.paid = false;
        loan.deadline = _deadline;
        loan.cancelled = false;

        emit LoanCreated(
            lastLoanId,
            loan.borrower,
            loan.token,
            loan.amount,
            loan.nftCollection,
            loan.nftId,
            loan.duration,
            loan.interestRate,
            loan.collateralValue,
            loan.deadline
        );
    }
```

Path: ./contracts/contract.sol : requestLoan(), acceptLoan()

Recommendation: It is recommended to perform additional validation of the lending token within the *acceptLoan()* function.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: The lending token validation is now performed within the *acceptLoan()* function as well.

L02. Solution Does Not Support Non-standard ERC721 Tokens

Impact	Low
Likelihood	Low

The Lending solution allows for borrowing the lending token with the collateral in the form of a valuable NFT token. The NFT token is transferred to the Lending contract within the *acceptLoan()* function. This functionality assumes that the selected NFT token implements OpenZepplin's *IERC721* interface and the *safeTransferFrom()* function in particular, which might not necessarily be true. CryptoPunks and CryptoKitties are examples of non-standard NFT tokens that do not support this particular *IERC721* interface. Therefore, such NFTs cannot be used as a collateral within the Lending solution.

```
function acceptLoan(uint256 _loanId) external nonReentrant {
        Loan storage loan = loans[_loanId];

        require(loan.borrower != address(0) && loan.lender == address(0), "Lending:
invalid loan id");
        require(!loan.cancelled, "Lending: loan cancelled");
        require(loan.deadline > block.timestamp, "Lending: loan acceptance deadline
passed");

        loan.lender = msg.sender;
        loan.startTime = block.timestamp;

        IERC20(loan.token).safeTransferFrom(msg.sender, loan.borrower,
loan.amount);
        IERC721(loan.nftCollection).safeTransferFrom(loan.borrower, address(this),
loan.nftId);

        emit LoanAccepted(_loanId, loan.lender, loan.startTime);
    }
```

Path: ./contracts/contract.sol : repayLoan(), acceptLoan(), claimNFT(), liquidateLoan()

Recommendation: It is recommended to consider implementing ERC721 transfer mechanism that covers a broader range of available NFTs.

Found in: 6460ac1

Status: Fixed (Revised commit: 7c2e10)

Remediation: The *requestLoan()* function now implements additional assertion that enforces the NFT collateral to implement the IERC721 interface, to ensure that it is compliant with ERC721.

L03. OnERC721Received Hook Does Not Check Token Ownership

Impact	Low

Likelihood	Low

The Lending solution allows for borrowing the lending token with the collateral in the form of a valuable NFT token. The NFT token is transferred to the Lending contract within the *acceptLoan()* function. This functionality assumes that the selected NFT token implements OpenZepplin's *IERC721* interface and the *safeTransferFrom()* function, which requires implementation of the *onERC721Received()* hook, if the recipient is a smart contract.

The Lending contract implements the *onERC721Received()* hook, however it does not process the hook's input data, to verify whether the NFT token was indeed transferred by checking the ownership.

```
   function onERC721Received(address, address, uint256, bytes calldata) external
pure returns (bytes4) {
        return IERC721Receiver.onERC721Received.selector;
   }
```

Path: ./contracts/contract.sol: onERC721Received()

Recommendation: It is recommended to verify whether upon calling the hook the NFT with provided token Id is indeed owned by the Lending contract.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: The *requestLoan()* function now implements additional assertion that enforces the NFT collateral to implement the IERC721 interface, to ensure that it is compliant with ERC721. Checking the ownership of NFT by means of the *ownerOf()* function was considered not optimal, as it relies on another ERC721 functionality.

L04. FeeReductionFactor Can Be Set to 0

Impact	Medium
Likelihood	Low

The Lending solution applies multiple various fees during loan repayment including *OriginationFee*. The *OriginationFee* is scaled based on the *OriginationFeeRanges* and the *feeReductionFactor*. However, it was identified that *feeReductionFactor* lacks any input validation and can be mistakenly set to 0 value. In case of such an event, the *getOriginationFee()* function call may revert due to an attempt of division by zero operation. The *getOriginationFee()*

function is used directly by the *repayLoan()* and *liquidateLoan()* functions. Eventually, it may prevent loan repayment or liquidation.

```
function setFeeReductionFactor(uint256 _factor) external onlyOwner {
    feeReductionFactor = _factor;
}
```

```
function getOriginationFee(uint256 _amount) public view returns (uint256) {
    uint256 originationFee = baseOriginationFee;

    for (uint256 i = 0; i < originationFeeRanges.length; i++) {
        if (_amount < originationFeeRanges[i]) {
            break;
        } else {
            originationFee = (originationFee * PRECISION) / feeReductionFactor;
        }
    }
    return (_amount * originationFee) / 100 / PRECISION;
}
```

Path: ./contracts/contract.sol: setFeeReductionFactor()

Recommendation: It is recommended to implement input validation within the *setFeeReductionFactor()* function.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: The fee reduction factor cannot be set now below the *PRECISION* constant, which is equal to 10000.

L05. OriginationFeeRanges Collection Lacks Input Validation

Impact	Medium
Likelihood	Low

The Lending solution applies multiple various fees during loan repayment including *OriginationFee*. The *OriginationFee* is scaled based on the *OriginationFeeRanges* and the *feeReductionFactor*. However, it was identified that the OriginationFeeRanges collection lacks any input validation and can be mistakenly set to incorrect values. In case of such an event, the *getOriginationFee()* function call may return an incorrect value, e.g. without application of a reduction factor.

```
function setRanges(uint256[] memory _originationFeeRanges) public onlyOwner {
        originationFeeRanges = _originationFeeRanges;
    }
```

```
function getOriginationFee(uint256 _amount) public view returns (uint256) {
        uint256 originationFee = baseOriginationFee;

        for (uint256 i = 0; i < originationFeeRanges.length; i++) {
            if (_amount < originationFeeRanges[i]) {
                break;
            } else {
                originationFee = (originationFee * PRECISION) / feeReductionFactor;
            }
        }
        return (_amount * originationFee) / 100 / PRECISION;
    }
```

Path: ./contracts/contract.sol: setRanges()

Recommendation: It is recommended to implement input validation within the *setRanges()* function, such that the provided values must be in ascending order.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: The origination fee ranges have now implemented input validation. At least one record must be provided, but no more than six. The record must be non-zero value. Values must be provided in ascending order.

L06. ERC20 Zero Amount Transfers Possible

Impact	Low
Likelihood	Low

The Lending solution applies multiple various fees during loan repayment or liquidation. When transferring fees, it is possible to have fees equal to zero, causing unnecessary ERC20 transfers to be executed.

The fee can be equal to zero if the combination of following conditions are true:

- The protocol fee is set to 0.
- The grace fee is set to 0.
- The base origination fee is set to 0.
- The liquidation fee is set to 0.

Although ERC20 zero amount transfers are usually possible, it does not add any value to the protocol.

Path: ./src/Lending.sol : repayLoan(), liquidateLoan().

Recommendation: It is recommended to check whether the fee is greater than zero before transfer and skip it otherwise.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: The platform fee transfer is now skipped if it is equal to zero.

Informational

I01. NFT Tokens Are Not Whitelisted

The Lending solution allows for borrowing the lending token with the collateral in the form of a valuable NFT token. The lending token must be firstly whitelisted by the solution owner. In contrast, the NFT token used for collateral is not whitelisted. Still, to use any NFT it must be firstly evaluated and pre-set within the *IPriceIndex* contract. Thus, it can be considered as indirect whitelisting, however, this configuration depends on the third party smart contract, which can be considered as a deviation from the leading security standards.

Path: ./contracts/contract.sol : requestLoan()

Recommendation: It is recommended to consider implementation of NFT tokens whitelisting that can be used within the Lending solution.

Found in: 6460ac1

Status: **Mitigated** (Revised commit: 7c2e10)

Remediation: The solution now implements the NFT blacklisting. The solution owner can disallow the use of a particular NFT from collection as a collateral. Implementing blacklisting over whitelisting is considered a deviation from the leading security standard, thus, the finding is perceived as *Mitigated*.

I02. Inconsistent Usage of PRBMath Library

The Lending solution allows lender to lend an amount of tokens to the borrowers, which should be paid off within the time of loan duration or grace period. During this time the interest is accrued and various fees are applied. To calculate the debt amount the

getDebtWithPenalty() function is used, which uses PRBMath's *UD60x18* struct and related functions for calculations.

```
function getDebtWithPenalty(
        uint256 _borrowedAmount,
        uint256 _apr,
        uint256 _loanDuration,
        uint256 _repaymentDuration
    ) public pure returns (uint256) {
        if (_repaymentDuration > _loanDuration) {
            _repaymentDuration = _loanDuration;
        }
        UD60x18 accruedDebt = convert((_borrowedAmount * _apr * _repaymentDuration)
/ SECONDS_IN_YEAR / 100 / PRECISION);
        UD60x18 penaltyFactor = convert(_loanDuration -
_repaymentDuration).div(convert(_loanDuration));

        return convert(accruedDebt.add(accruedDebt.mul(penaltyFactor)));
    }
```

In contrast, *getOriginationFee()*, *getLiquidationFee()* and grace fee do not use any additional math library for calculations.

```
function getLiquidationFee(uint256 _borrowedAmount) public view returns (uint256)
{
        return (_borrowedAmount * liquidationFee) / PRECISION;
    }
```

No specific vulnerability was identified due to this fact, however, the finding was reported to drag attention of the development team to the subject matter.

Path: ./contracts/contract.sol: getDebtWithPenalty(), getOriginationFee(), getLiquidationFee(), repayLoan().

Recommendation: It is recommended to consider implementing the usage of PRBMath's *UD60x18* struct and related functions across the solution.

Found in: 6460ac1

Status: **Mitigated** (Revised commit: 7c2e10)

Remediation: Both the *getDebtWithPenalty()* function and the *getOriginationFee()* function are using *UD60x18* struct for arithmetic operations. The *getLiquidationFee()* function and grace fee still do not use it.

I03. Floating Pragma

The Lending solution has floating pragma set to *^0.8.19*.

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the *pragma* helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Paths: ./src/Lending.sol; ./src/IPriceIndex.sol.

Recommendation: It is recommended to lock the pragma version in all contracts to one of the newest compiler versions, considering all publicly known bugs related to such version.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: The pragma version is now locked on 0.8.19.

I04. Solidity Style Guide Violation

Contract readability and code quality are influenced significantly by adherence to established style guidelines. In Solidity programming, there exist certain norms for code arrangement and ordering. These guidelines help to maintain a consistent structure across different contracts, libraries, or interfaces, making it easier for developers and auditors to understand and interact with the code.

The suggested order of elements within each contract, library, or interface is as follows:

- Type declarations
- State variables
- Events
- Modifiers
- Functions

Functions should be ordered and grouped by their visibility as follows:

- Constructor
- Receive function (if exists)
- Fallback function (if exists)
- External functions
- Public functions
- Internal functions
- Private functions

Within each grouping, view and pure functions should be placed at the end.

Furthermore, following the Solidity naming convention and adding NatSpec annotations for all functions are strongly recommended. These measures aid in the comprehension of code and enhance overall code quality.

Path: ./src/Lending.sol.

Recommendation: Consistent adherence to the official Solidity style guide is recommended. This enhances the readability and maintainability of the code, facilitating seamless interaction with the contracts. Providing comprehensive NatSpec annotations for functions and following Solidity's naming conventions further enrich the quality of the code.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: The code is now rearranged to adherence the style guidelines.

I05. Functions that Should Be External

Public functions that are not called from inside the contract should be declared external.

Path: ./src/Lending.sol : setRanges().

Recommendation: Consider changing the function visibility to external.

Found in: 6460ac1

Status: **Fixed** (Revised commit: 7c2e10)

Remediation: The aforementioned function is now set to external.

I06. Constant Accuracy Mismatch

The *SECONDS_IN_YEAR* constant is set to 360 days instead of 365, which might have an unintentional impact on the interest calculations.

```
uint256 private constant SECONDS_IN_DAY = 3600 * 24;
```

Path: ./src/Lending.sol: SECONDS_IN_YEAR

Recommendation: It is recommended to consider setting the *SECONDS_IN_YEAR* constant to 365 instead of 360 days.

Found in: 6460ac1

Status: **Fixed**

Remediation: The client's team uses 360 valu intentionally. It is explained in the FAQ that the Annual Percentage Rates (APR) is calculated on a 30/360 year basis.

I07. Gas Optimisation Possible in for Loops

In Solidity version 0.8 and above, arithmetic operations automatically include checks for underflows and overflows. Although these checks are useful for preventing calculation errors, they consume additional Gas, leading to higher transaction costs.

In scenarios where underflows and overflows are not possible, the additional checks introduced by Solidity 0.8 can be bypassed to save Gas. This can be done by placing the increment or pre-increment operation inside an *unchecked{}* block. This block enables developers to perform arithmetic operations without the automatic underflow and overflow checks, thus conserving Gas when they are not needed.

It is a well-known fact that pre-increment *++i* costs less Gas than increment *i++* operator.

Additionally, the loops in the provided code snippets read the length of the array stored in storage, which is expensive as opposed to the situation when the length is once set to the memory variable and read from it every single loop iteration.

```
function _setLoanTypes(uint256[] memory _durations, uint256[] memory
_interestRates) internal {
       require(_durations.length == _interestRates.length, "Lending: invalid
input");
       for (uint256 i = 0; i < _durations.length; i++) {
           require(_interestRates[i] <= MAX_INTEREST_RATE, "Lending: cannot be
more than max");
           aprFromDuration[_durations[i]] = _interestRates[i];
       }
   }
```

Path: ./src/Lending.sol: setTokens(), unsetTokens(), unsetLoanTypes(), getOriginationFee(), _setLoanTypes().

Recommendation: It is recommended to optimize the code by creating memory variables to store the length of the loop.

It is recommended to use a pre-increment operator inside an *unchecked{}* block for Gas optimization inside loops.

Found in: 6460ac1

Status: Fixed (Revised commit: 7c2e10)

Remediation: All for loops are now optimized in terms of Gas consumption.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	**Medium**
Medium Likelihood	**High**	**Medium**	**Low**
Low Likelihood	**Medium**	**Low**	**Low**

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/lucidao-developer/altr-lending-smart-contracts
Commit	6460ac162c704536adda75151f177cba22b7aa3c
Whitepaper	Link
Requirements	Link
Technical Requirements	Link
Contracts	File: IPriceIndex.sol SHA3: 40f7cefdf63d2d1349397fc38259aee0d248693f09d48c4f4b9c73acc17abb45 File: Lending.sol SHA3: 2fed985cb1e63c7b8731057fb9fd7dbca32fe0393182e719b1c0ea3990fc352c

Second review scope

Repository	https://github.com/lucidao-developer/altr-lending-smart-contracts
Commits	7c2e10859755e1708e0c50beedbe3f8e77b845f5, b1af058882db27d270eac22d69361a01ab0795e2
Whitepaper	Link
Requirements	Link
Technical Requirements	Link
Contracts	7c2e108: File: IAllowList.sol SHA3: 1fe2204d6da4f730a3d02c191c8de859404db463ef69a5fb2ebd267bb413271c File: IPriceIndex.sol SHA3: 01e616deecb7ed5a2874f008e21e5e524ad93a109457ac4b268d0cb795907fe3 File: Lending.sol SHA3: 8108ece86068f784af85b12310b8803421ba31d0c75faa60289eb3cb8fc053d8 b1af05: File: Lending.sol SHA3: 91686010c6478c5d53bdafdc65e998840c10d42ec8a87c61e38c5f4225755aff