QuillAudits

Audit Report
December, 2021

For

Nord NFT Loans

Contents

Overview

NordLoan by Nord Finance

NordLoan provides functionality to borrow the Loan on NFT as collateral. Repay the Loan and liquidate the loan if not repaid on time.

Scope of the Audit

The scope of this audit was to analyse **NordLoan smart contract**'s codebase for quality, security, and correctness.

NordLoan Contract: https://github.com/nordfinance/nord-nft-loans/blob/development/contracts/NordLoan.sol
Branch: Development
Commit: 39b2230d850e0fa4eea00a5dc19e58eb86a1605f
Fixed In: 07b62e65f9abafef9e9f28d1ed3c10b2f689fd98

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC20 transfer() does not return boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Mythril, Slither, SmartCheck, Surya, Solhint.

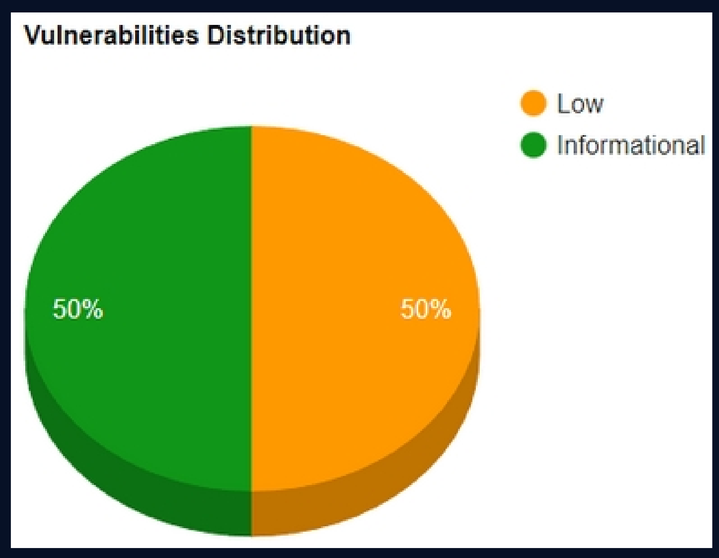
Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

Risk-level	Description
High	A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.
Medium	The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.
Low	Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.
Informational	These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

Type	High	Medium	Low	Informational
Open	0	0	0	0
Acknowledged	0	0	1	1
Closed	0	1	1	4



Functional Testing Results

Complete functional testing report has been attached below:
NordLoan TestCases.

Some of the tests performed are mentioned below:

- should be able to borrow
- should revert if repayAmount is less than principalAmount
- should revert if loan duration is greater than max loan duration
- should revert if loan duration is zero
- should revert if adminFee for ERC20 token is zero
- should revert if admin fee has changed
- should revert if ERC20 is not whitelisted
- should revert if NFT contract is not whitelisted
- should revert if borrower nonce is already used
- should revert if lender nonce is already used
- should revert if incorrect borrower signature is used
- should revert if incorrect lender signature is used
- should be able to payback loan
- should revert on payback of an already repaid/liquidated loan
- should revert on invalid loanId
- should be able to liquidate overdue loan
- should revert on loans not overdue
- should revert on signature reuse
- should revert on use of other user's signature

Issues Found

High severity issues

No issues were found.

Medium severity issues

1. **Failing low-level call**

   [#L594] function **_attemptTransferFrom** performs low-level **call** operation over NFT contract involved for:
   - Approving contract address itself as the spender of NFT Id, which it is already an owner of
   - Transferring NFT Id from the contract address to the recipient.

```
594     function _attemptTransferFrom(
595         address _nftContract,
596         uint256 _nftId,
597         address _recipient
598     ) internal returns (bool) {
599         _nftContract.call(
600             abi.encodeWithSelector(
601                 IERC721(_nftContract).approve.selector,
602                 address(this),
603                 _nftId
604             )
605         );
```

However, logically, there is no need for an owner of an NFT Id to approve itself as the spender of it. Also, if the NFT contract follows the Openzeppelin's ERC721 implementation, the first low-level call to approve will always fail, as OZ's ERC721 implementation does not allow it.

```
112     function approve(address to, uint256 tokenId) public virtual override {
113         address owner = ERC721.ownerOf(tokenId);
114         require(to != owner, "ERC721: approval to current owner");
115
116         require(
117             _msgSender() == owner || isApprovedForAll(owner, _msgSender()),
118             "ERC721: approve caller is not owner nor approved for all"
119         );
120
121         _approve(to, tokenId);
122     }
```

Reference: OZ's ERC721 ***approve*** *function*

Recommendation: Consider reviewing and verifying the operational and business logic

Status: Fixed

Low severity issues

2. Missing return value checks for ERC20 transfer operations

The functions ignore the return value for ERC20 transfer operations. Although the tokens following OpenZeppelin ERC20 implementation will work correctly, as they **revert** on **failure** and return **true** on **success**. But there exist many such tokens which behave differently, as it's nowhere defined that the ERC-20 contract has to **revert** on **failure**. One such variation is **0x**'s ZRX Token, which returns **false** on failure.

As a consequence, for such variation if the transfer fails, the **beginLoan** operation will still succeed. As a result the borrower will still lose its NFT as collateral but not receive the ERC20 tokens as agreed loan.

The scenario can be created intentionally by ProtocolAdmin, by whitelisting a malicious ERC20 contract.

Recommendation:
- Use SafeERC20 wrapper provided by OpenZeppelin for ERC20 operations
- Ensure that no malicious contracts are added by Protocol Admin as whitelisted contracts

References:
- https://soliditydeveloper.com/safe-erc20
- https://twitter.com/Uniswap/status/1072286773554876416
- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol

Status: Fixed

3. Required Zero-Trust Policy

As the crucial aspect of the NordLoan system is the offline agreement and signature of borrower and lender. If the attacker/borrower succeeds in convincing a lender to sign a malformed message hash(which contains the borrower's desired parameters) with the help of phishing or social-engineering attacks, it may allow the borrower to take a loan with any desired parameters, hence it is necessary and required that every lender should follow a zero-trust policy and sign their message hashes by themselves.

Recommendation: Notify and announce the need of zero-trust policy to the lenders.

Status: Acknowledged

Informational issues

4. Public functions that are never called by the contract should be declared external to save gas.

The functions are:
- beginLoan
- getPayoffAmount
- getWhetherNonceHasBeenUsedForUser

Status: Fixed

5. **Multiple pragma directives have been used.**

Recommendation:
Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using ^ in pragma solidity 0.5.10) ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs. Ref: Security Pitfall 2

Status: **Fixed**

6. Comparison with boolean constants

```
341             // Calculate amounts to send to lender and admins
342             uint256 interestDue = loan.maximumRepaymentAmount -
343                 loan.loanPrincipalAmount;
344             if (loan.interestIsProRated == true) {
345                 interestDue = _computeInterestDue(
346                     loan.loanPrincipalAmount,
347                     loan.maximumRepaymentAmount,
```

```
479     //          increased slightly.
480     function getPayoffAmount(uint256 _loanId) public view returns (uint256) {
481         Loan storage loan = loanIdToLoan[_loanId];
482         if (loan.interestIsProRated == false) {
483             return loan.maximumRepaymentAmount;
484         } else {
```

Recommendation:
Boolean constants can be used directly and do not need to be compared to true or false

Status: **Fixed**

7. **Inefficient check**

```
191         );
192         require(
193             uint256(loan.loanAdminFeeInBasisPoints) ==
194                 adminFeeInBasisPoints[_loanERC20Denomination],
195             "The admin fee has changed since this order was signed."
196         );
197
```

[#L140] function **beginLoan** adds a check to make sure that the input value **loan.loanAdminFeeInBasisPoints** should be strictly equal to **adminFeeInBasisPoints** of the desired **_loanERC20Denomination** contract set by **ProtocolAdmin**.

However, as it checks for a specific value for the parameter, there is no need to ask it as an input from the borrower, instead the admin fee for the particular ERC20 contract may be directly assigned to **loanAdminFeeInBasisPoints** in the loan struct.

Recommendation:
Consider reviewing and verifying the operational and business logic

Status: Acknowledged

8. **Unnecessary use of contract address to generate function signature**

[#L601, #L608, #L624] function **_attemptTransferFrom** and **_attemptTransfer** creates function signature from interface but involves the contract address as well.

For reference:

```
606             (bool success, ) = _nftContract.call(
607                 abi.encodeWithSelector(
608                     IERC721(_nftContract).transferFrom.selector,
609                     address(this),
610                     _recipient,
611                     _nftId
612                 )
613             );
```

However, the function signature can be directly generated by the interface itself. Hence, in this case **IERC721.transferFrom.selector w**ill also generate the same 4 byte signature.

Recommendation:
Consider avoiding the contract address for function signature creation.

Status: **Fixed**

Closing Summary

Some issues of Medium and Low severity were found, which are now fixed by Developers. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the **Nord**. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the **Nord** team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report
December, 2021

For

Nord NFT Loans

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com