#HashEx
BLOCKCHAIN SECURITY

NFTY Loans

smart contracts final audit report

January 2023

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the NFTY team to perform an audit of their smart contracts. The audit was conducted between 2022-12-26 and 2022-12-28.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at @NFTYNetwork/nftyfinance-contracts GitHub repository after the 6338193 commit.

The NftyLending contract is designed to be deployed behind a proxy. Users need to check the current system admin before interacting with the contacts.

Update. The NFTY team has responded to this report, the updated code is located in the same repository after the bd28b55 commit.

2.1 Summary

Project name	NFTY Loans
URL	https://nftynetwork.io
Platform	Binance Smart Chain
Language	Solidity

2.2 Contracts

Name	Address
SmartNft	
DIAOracleV2	
NftyLending	
Interfaces contracts	

3. Found issues

High	3 (16%)
Medium	4 (21%)
Low	3 (16%)
Info	9 (47%)

C1. SmartNft

ID	Severity	Title	Status
C1-01	Low	Gas optimizations	Partially fixed
C1-02	Info	Typos	Partially fixed

C2. DIAOracleV2

ID	Severity	Title	Status
C2-01	Medium	Overcentralization	Acknowledged
C2-02	Info	No visibility is specified	Acknowledged

C3. NftyLending

ID	Severity	Title	Status
C3-01	High	Upgradeability of a contract holding users' funds	Acknowledged
C3-02	High	Oracle data expiration check	Resolved
C3-03	High	Unlimited fee percent	Resolved
C3-04	Medium	Additional risk for shops with automated approvals	Acknowledged
C3-05	Medium	Owner can disable fees	Resolved
C3-06	Medium	Locked fees	Resolved
C3-07	Low	Gas optimizations	Partially fixed
C3-08	Info	Increased probability of hash collisions	Acknowledged
C3-09	Info	Reentrancy guards	Resolved
C3-10	Info	Typos	Resolved
C3-11	Info	Purpose of partial payback is unclear	Acknowledged
C3-12	Info	Price feeds collisions	Acknowledged
C3-13	Info	Not implemented code	Partially fixed

C4. Interfaces contracts

ID	Severity	Title	Status
C4-01	Low	Gas optimizations	Resolved
C4-02	Info	Typos	Resolved

4. Contracts

C1. SmartNft

Overview

An ERC-721 token standard implementation for promissory notes and obligation receipts of NftyLending.

Issues

C1-01 Gas optimizations

Low Partially fixed

1. Redundant code: `setLoanCoordinator()` duplicates the `grantRole()` function, `chainid` value is available without assembly (see `_getChainID()` function) since the 0.8.0 version of Solidity
2. Code for testing: import of Hardhat console should be removed before a mainnet launch.

C1-02 Typos

Info Partially fixed

Typos reduce the code's readability. Typos in 'SmarNFT', 'assigne', 'reciving'.

C2. DIAOracleV2

Overview

A simple oracle to be filled by the single owner address. Supports price feeds for multiple tokens and expirations timestamps.

Issues

C2-01 Overcentralization

Medium Acknowledged

A single address has full control over the oracle: it can update any prices for any timestamps (in the past or future) without any restrictions. Losing control over this account may lead to severe consequences for oracle clients.

Recommendation

We recommend securing the `oracleUpdater` account as much as possible and documenting actions taken.

C2-02 No visibility is specified

Info Acknowledged

The `oracleUpdater` has no visibility specified, the default value of `internal` is used. We recommend always explicitly specifying visibility even if it matches the default.

C3. NftyLending

Overview

Lending contract allowing user, a.k.a. shop owner, to create an NFT shop for whitelisted NFT collection and whitelisted ERC20 token, and other users to borrow ERC20 token of the shop with NFT as collateral for one of 3 available loan duration. Overdue loans are subject to liquidation.

Issues

C3-01 Upgradeability of a contract holding users' funds

High Acknowledged

The lending contract is designed to be deployed behind a proxy. The liquidity of the shops and NFTs used as collateral are held on this contract. If an attacker gets access to the proxy owner's account, they will gain control over these assets.

Recommendation

Consider using a non-upgradeable contract or secure the proxy owner's account with a MultiSig wallet.

C3-02 Oracle data expiration check

High Resolved

The `setFeeExpiration()` function is used for adjusting the `feeExpirationTime` parameter, which defines the timeframe of acceptable prices from oracle. The owner can set an arbitrary positive value, effectively disabling the oracle timestamps if `feeExpirationTime` is set to the distant future.

```
function setFeeExpiration(uint256 _feeExpirationTime) external onlyOwner {
    require(_feeExpirationTime > 0, "fee expiration cannot be 0");
    emit FeeExpirationSet(_feeExpirationTime);
}
```

Recommendation

Limit the updating value from above with a reasonable constant. Transfer the ownership to a Timelock contract with MultiSig admin.

C3-03 Unlimited fee percent

High Resolved

The `setLoanOriginationFees()` function is used to update the `loanOriginationFeePercentage`

variable, initially set to 1%, which defines the percent of the loan value to be split in form of NFTY tokens between lender, borrower, and platform. The owner can set an arbitrary percent, that may effectively block any interaction with the contract if it's set to a high value.

Recommendation

Limit the updating value from above with a reasonable constant. Transfer the ownership to a Timelock contract with MultiSig admin.

C3-04 Additional risk for shops with automated approvals

Medium Acknowledged

A token owner of an approved NFT as collateral may exploit the shops with automatic approvals of offers. Let's assume an approved NFT token had an open mint function. In that case, the attacker might mint as many NFTs as needed, use them as collateral and withdraw all liquidity from shops with automatic approvals.

Recommendation

Use a Merkle proof for freezing the ids of acceptable NFTs when an NFT is added as supported or at the moment of shop creation.

C3-05 Owner can disable fees

Medium Resolved

The `setPlatformFees()` function is used to update fees split between lender, borrower, and platform. The total fee percent is required to be 100%, but the owner can disable any 1 or 2 fee types, e.g. disable the shop's part of the fees.

```
function setPlatformFees(PlatformFees memory _platformFees) public onlyOwner {
    require(
        _platformFees.lenderPercentage.add(_platformFees.borrowerPercentage).add(_platformF
ees.platformPercentage) == 100,
        "fees do not add up to 100%" );
    platformFees.lenderPercentage = _platformFees.lenderPercentage;
    platformFees.borrowerPercentage = _platformFees.borrowerPercentage;
```

```
    platformFees.platformPercentage = _platformFees.platformPercentage;
    emit PlatformFeesSet(platformFees);
}
```

Recommendation

Add a minimal percentage for any of the 3 fees or document the owner's ability to disable some fees or fix fees individually upon the store creation.

C3-06 Locked fees

Medium Resolved

The escrow part of the fee is taken in the `_acceptOffer()` function and locked in the NftyLending contract without the possibility of withdrawing.

```
function _acceptOffer(Offer memory _offer, address _borrower) internal {
    ...
    IERC20Upgradeable(nftyTokenContract).safeTransferFrom(
        _borrower,
        address(this),
        borrowerFees.add(escrowFees)
}
```

Recommendation

Document this lock of NFTY tokens and add some form of tracking locked fees.

C3-07 Gas optimizations

Low Partially fixed

1. Unnecessary writing to storage: `liquidityShops` mapping from `id` to structure with the same `id`.
2. The `setErc20()` and `setNft()` functions don't allow removing token addresses with simultaneous clearance of `minimumBasketSize`, `minimumPaymentAmount`, and `image` fields.
3. Redundant code: the `getPriceIfNotOlderThan()` returned price is unnecessarily cast into uint256 type.

4. Multiple reads from storage: `whitelistedErc20s.length()` and `whitelistedErc20s.at()` in the `getWhitelistedErc20s()` function, `whitelistedNfts.length()` and `whitelistedNfts.at()` in the `getWhitelistedNfts()` function, `liquidityShops[id].owner` in the `getLiquidityShop()` function, `liquidityShop.owner` in the `addLiquidityToShop()` function, `liquidityShop.balance` and `liquidityShop.owner` in the `liquidityShopCashOut()` function, `liquidityShop.owner` in the `freezeLiquidityShop()` function, `liquidityShop.owner` in the `unfreezeLiquidityShop()` function, `promissoryNoteToken`, `loan.liquidityShopId`, `loan.nftCollateralId`, `loan.smartNftId`, and `obligationReceiptToken` in the `liquidateOverdueLoan()` function, `liquidityShop.balance`, `liquidityShop.owner`, and `loanIdCounter.current()` in the `_acceptOffer()` function, `erc20s[].minimumPaymentAmount`, `loan.remainder`, `loan.duration`, `loan.amount`, `loan.liquidityShopId`, `loan.smartNftId`, `obligationReceiptToken`, and `promissoryNoteToken` in the `payBackLoan()` function.
5. Unnecessary read from storage of variable that is already in memory: `platformFees` in the `setPlatformFees()` function, `shopIdCounter.current()` in the `createLiquidityShop()` function, `liquidityShop.id` and `liquidityShop.balance` in the `addLiquidityToShop()` function, `liquidityShop.id` in the `liquidityShopCashOut()` function, `liquidityShop.id` in the `freezeLiquidityShop()` function, `liquidityShop.id` in the `unfreezeLiquidityShop()` function, `liquidityShop.id` in the `_acceptOffer()` function, `loan.smartNftId` and `loan.remainder` in the `payBackLoan()` function.
6. The `memory` keyword for `loan` struct reading is gas-wise in `liquidateOverdueLoan()` and `payBackLoan()` functions since most of their fields are read multiple times.
7. The `storage` keyword for loan struct reading is gas-wise in `createLoan()` and `acceptOffer()` functions since only a single structure field is used.
8. The `_storeLoan()` function should calculate the loan end-time instead of storing duration in order to save gas on loan payback or liquidation.
9. The `_storeLoan()` should take `loanIdCounter.current()` counter from parameters since it's already read in the `_acceptOffer()` function.

C3-08 Increased probability of hash collisions

● Info ⊘ Acknowledged

The `_acceptOffer()` function uses truncated `keccak256` hash as id for minting `promissoryNoteToken` and `obligationReceiptToken` NFTs. Truncation increases the probability of hash collisions, which may lead to a temporary halt of the contract.

C3-09 Reentrancy guards

Info Resolved

There are a lot of interactions with the external contracts in the contract's functions. Also, the tokens may have callbacks on transfers, which makes it possible to perform reentrancy attacks. Although the current version of the code is not vulnerable to reentrancy attacks, taking in mind the upgradability of the contracts, we recommend adding reentrancy guards beforehand to ensure safety in code updates.

C3-10 Typos

Info Resolved

Typos reduce the code's readability. Typos in 'chekcs', 'boolian'.

C3-11 Purpose of partial payback is unclear

Info Acknowledged

The `payBackLoan()` function allows the borrower to pay back the loan partially, resulting only in the `PaymentMade` event being emitted. Excessive gas could be spent if a borrower mistakenly set a smaller payback amount.

C3-12 Price feeds collisions

Info Acknowledged

The `getOfferFees()` function calls for `getPriceIfNotOlderThan()` function with the concatenated string of token symbols. If 2 or more ERC20 tokens were added to the whitelist of allowed tokens, the oracle data would be shared across these tokens.

```
function getOfferFees(uint256 amount, address currency) internal view returns (uint256){
    ...
    getPriceIfNotOlderThan(
        string(abi.encodePacked(IERC20Metadata(currency).symbol(), "/USD")),
        uint128(feeExpirationTime) );
    ...
}
```

Recommendation

We recommend avoiding mapping `string => uint256` in the DIAOracleV2 in favor of `bytes32 => uint256`, the `bytes32` would allow hashing any type of data, e.g. token symbols + token address.

C3-13 Not implemented code

Info Partially fixed

The `createLiquidityShop()` function is used to add a new shop with `LiquidityShop` struct parameters. Boolean `allowRefinancingTerms` parameter of the shop is described as 'Whether or not this liquidity shop will accept refinancing terms', but this feature is not implemented anywhere in the reviewed contracts. The shop loans can be only paid back or liquidated, other options aren't presented.

Recommendation

Since the contract is intended to be deployed with a proxy, this unimplemented feature can be justified, but it needs to be explicitly described.

C4. Interfaces contracts

Overview

Interfaces for DIAOracleV2 and NftyLending contracts.

Issues

C4-01 Gas optimizations

Low Resolved

1. The `LiquidityShop` structure in the INftyLending interface contains `id` shop identifier but the NftyLending contract has a mapping in the form of `id => LiquidityShop`, storing the `id` value twice.

2. The `LiquidityShop` structure in the INftyLending interface could be rearranged for compact packing: boolean variables should be placed adjacent to the address to fit into a single 32-bytes slot.

C4-02 Typos

Info Resolved

Typos reduce the code's readability. Typo in INftyLending interface in 'liqudity'.

5. Conclusion

3 high, 4 medium, 3 low severity issues were found during the audit. 2 high, 2 medium, 1 low issues were resolved in the update.

The reviewed contracts are highly dependent on the owner’s account. Some of the reviewed contracts are designed to be deployed behind proxies. Users using the project have to trust the owner and that the owner's account is properly secured and need to check the current system admin before interacting with the contacts.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY