PUBLIC

Code Assessment

of the NFTfi Marketplace Smart Contracts

October 08, 2021

Produced for

by

Contents

1 Executive Summary

Dear NFTfi team,

First and foremost we would like to thank NFTfi for giving us the opportunity to assess the current state of their NFTfi Marketplace system. This document outlines the findings, limitations, and methodology of our assessment.

All identified findings have been resolved.

The communication with the team was helpful in resolving open questions.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	5
• Code Corrected	5
Low-Severity Findings	11
• Code Corrected	10
• Specification Changed	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the NFTfi Marketplace repository based on the documentation files. The smart contracts in the `contracts` directory were assessed. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	August 27 2021	40235826e735f381e4dfeb367544365c4b748c22	Initial Version
2	October 07 2021	cde92876065456ab7da30bbdb054798df39f196b	Second Version

For the solidity smart contracts, the compiler version `0.8.4` was chosen.

2.1.1 Excluded from scope

The following smart contracts are excluded from the scope:

- `contracts/governance`
- `contracts/test`

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

NFTfi offers a platform for receiving loans while offering an NFT as collateral. The current system supports peer-to-peer loans. Meaning, a borrower offers an NFT as collateral and a lender transfers ERC-20 tokens to the borrower. In case the payback time is exceeded, the lender has the right to liquidate the loan and withdraw the NFT from the lending contract. Furthermore, renegotiations of the loan terms are possible in the NFTfi Marketplace.

2.2.1 Loans

There are two ways of creating a loan:

- **The lender accepts a listing.**
    1. The lender sees the request for a loan in the front-end with the signed loan terms defined by the borrower.
    2. The lender calls `acceptListing` and provides an offer that matches the loan terms.
    3. The signature is verified.
    4. The loan is created. Hence, the NFT is locked in the loan contract while the ERC-20 tokens are transferred to the borrower.
- **The borrower accepts an offer.**

1. The borrower sees a loan offer, signed by the lender, in the front-end.
2. The borrower calls `acceptOffer` while providing the signed offer as an argument.
3. The signature is verified.
4. The loan is created. Hence, the NFT is locked in the loan contract while the ERC-20 tokens are transferred to the borrower.

The borrower and the lender receive receipts, the obligatory note and the promissory note. These are defined in the SmartNft contracts, are NFTs and, thus, are transferrable. The owner of the obligatory note receives the collateral back on payback. The owner of the promissory note receives the payment on payback or receives the NFT collateral on liquidation.

There are two ways to end a loan:

- The loan is paid back: The holder of the obligatory note (borrower) receives the NFT while the lender receives the amount lent plus the interest. Paybacks can occur at any time.
- The loan is liquidated: The holder of the promissory note (lender) receives the NFT. Liquidation can only occur if the loan repayment is overdue.

Furthermore, loans can be renegotiated at any time, even if the loan is overdue. The renegotiation process works as follows:

1. The lender signs a renegotiation offer and sends it to the borrower.
2. The borrower can accept by calling `renegotiateLoan`.
3. The signature is verified and the loan data is modified.

The current system supports two loan types:

- Fixed direct loans: Peer-to-peer loans with a fixed payback amount.
- Pro-rated direct loans: Peer-to-peer loans with an increasing payback amount. Early paybacks are cheaper than late paybacks.

Similar loan types (e.g. direct loans) share a loan coordinator who registers loans and manages the SmartNft minting and burning. Note that on loan creation a referral fee, specified by the borrower, could be transferred to a referrer, specified by the lender. Also, on token payback, the borrower pays an admin fee to the system. That fee is distributed to the governance and the revenue-share partner specified on loan creation by the borrower.

2.2.2 Supported Tokens

The system supports ERC-721 tokens, CryptoKitties and ERC-1155 tokens. Furthermore, the system allows bundles in form of ERC-998 top-down ERC-721 token bundles as collateral. The standard, however, was extended to also support ERC-1155 tokens in the bundles. However, the extended ERC-998 contract is not whitelisted but the ImmutableBundle contract address is. That contract wraps the bundle so that children cannot be modified during the loan-taking.

The NFT contracts whitelisted are stored in the PermittedNFTs contract. Similarly, supported ERC-20 tokens are whitelisted in the PermittedERC20s contract. NFTs contracts map also to transfer wrappers.

2.2.3 Registries

Besides the whitelisted tokens and the wrapper registry, the system has several other registries:

- PermittedPartners: Maps addresses to revenue shares for the admin fee.
- LoanRegistry: Maps loan types to contract addresses.
- NftfiHub: Central source of truth that provides the addresses for registries and loan coordinator addresses.

The system can be extended by deploying additional contracts and registering them appropriately in the system.

2.2.4 Roles and Trust Model

Users: There are two types of users, the borrowers and the lenders. Users of the system are untrusted and are expected to behave unpredictably.

Governance: The governance is expected to be the NFTfi DAO and is assumed to behave honestly. However, some safeguards to prevent errors should be in place. Also, the lending terms agreed on should always be honoured and should not be changeable by governance (e.g. fees agreed on creation, token support for payback, allowed payback/liquidations when the contract is paused). Governance is also trusted to carefully evaluate whitelisting and other decisions.

Referrer: The referrers receive fees and do not interact with the system directly. However, they are untrusted.

Revenue Share Partner: The revenue share partners are fully trusted and must be whitelisted by the governance.

Tokens: Tokens are trusted since governance must whitelist them.

2.2.5 Updates in Version 2

Some of the updates are mentioned in the following list, however this includes only some relevant additional functionality.

- Renegotiations also transfer fees to the governance.
- Airdropped ERC-721, ERC-1155 and ERC-20 tokens can now be drained by the governance.
- SmartNfts can only be held by EOAs or whitelisted contract addresses.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved, have been moved to the Resolved Findings section. All of the findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- Offer/Listing Signature Valid for Any Loan Contract Code Corrected

Medium-Severity Findings	5

- Broken/Partial ERC165 Support Code Corrected
- No Sanity Check on Revenue Share Code Corrected
- Renegotiation Replays Possible Code Corrected
- SmartNFTs May Not Be Composable With Other Protocols Code Corrected
- Undeployable SmartNFTs Code Corrected

Low-Severity Findings	11

- Anyone Can Liquidate Code Corrected
- Double Getters Code Corrected
- Event Issues Code Corrected
- Gas Inefficiencies Code Corrected
- Gas Inefficiencies in SmartNFTs supportsInterface() Code Corrected
- Maximum Number of Loans May Be Violated Code Corrected
- Maximum Repayment Amount Code Corrected
- Not Using safeTransfer for ERC-20 Transfers Code Corrected
- Renegotiation on Wrong Contract Possible Code Corrected
- Repetitive Validation on Batch Child Transfer Code Corrected
- Specification Mismatch Specification Changed

6.1 Offer/Listing Signature Valid for Any Loan Contract

Security High Version 1 Code Corrected

The current system requires the Offer or the ListingTerms struct to be signed. However, no information about for which contract address this struct is intended to be used. Thus, it allows the accepting party publishing a loan on-chain to decide whether the loan will be pro-rated or fixed (or other loan types in the future).

Assume the following scenario:

1. Alice talks off-chain to Bob and makes her the offer to lend 100 DAI for her ERC-721 collateral token as a fixed loan where the maximum repayment amount is 200 DAI. Bob sets the loan interest rate to 0 since it is not needed for the loan.
2. Bob sends the signed offer to Alice.
3. Alice now calls `acceptOffer` on the pro-rated contract. Loan terms get prepared for later calculations. `_acceptOffer` is called internally.
4. Now lender signatures are verified for the offer. Since nothing changed, the call succeeds.
5. Alice can pay back the loan cheaper than Bob agreed to.

In this scenario, Bob would need to have had the pro-rated contract approved for some DAI (e.g. Bob could be an active lender).

Furthermore, the signature could be used on multiple contracts. However, this requires the NFT to be a ERC-1155 token. In such a scenario, Alice could receive a pro-rated and a fixed loan while having only one signature of Bob.

Similar issues may arise in the case of signing listings where the lender could for example make a pro-rated loan a fixed loan. Since the borrower could be an active user of the platform, making both lending contracts an operator is a plausible assumption. Again, double-loans can be created if the NFT is an ERC-1155 token (assuming the contracts are operators of the user's NFTs). Especially, this is dangerous, since the documentation describes giving default NFT approvals for NFTfi contracts.

In conclusion, the system is unaware for which contract a signature is intended to be used. Nonces can be reused since they are not stored globally but per lending contract. Hence, replay attacks are possible.

Code corrected:

Now, the contract address is signed by the party signing. Thus, a loan can be only created on the intended contract.

6.2 Broken/Partial ERC165 Support

Design | Medium | Version 1 | Code Corrected

Through inheritance the contracts in the `composable` directory inherit from `ERC165` which implements EIP-165 that defines a standard method for publishing and detecting supported interfaces.

```
function supportsInterface(bytes4 interfaceID) external view returns (bool);
```

Not all of the aforementioned contracts do overwrite this function to extend its extended functionality.

- ERC9981155Extension additionally implements the IERC998ERC1155TopDown and the IERC1155Receiver interface.
- NftfiBundler implements the INftiBundler functions while it does not explicitly implement the interface (however, the naming suggests otherwise).
- ImmutableBundle further implements the IERC721Receiver interface.

Hence, `supportsInterface()` will not return `true` for some of the `interfaceId` it supports.

Code corrected:

The `supportsInterface` return `true` for `interfaceId` of all the implemented interfaces.

6.3 No Sanity Check on Revenue Share

Correctness Medium Version 1 Code Corrected

Partners can earn a share of the administrator fee. The percentage can be set with `PermittedPartners.setPartnerRevenueShare()`. However, this method does not check whether the share exceeds 100%.

Assume a loan starts where that is the case. Then, this percentage would be stored in the loan extras of a loan which cannot be renegotiated nor modified in any way for the loan. Paying back the loan will ultimately revert since an underflow would occur when computing the fee left for the administrator. Hence, the borrower cannot retrieve his collateral back and liquidation is the only possibility to exit the loan.

Code corrected:

100% cannot be exceeded anymore.

6.4 Renegotiation Replays Possible

Security Medium Version 1 Code Corrected

Renegotiation is a feature that allows the lender to give the borrower an alternative offer after the loan has been created. However, replay attacks may be possible here.

As more loan types will appear, more loan coordinator contracts could be deployed. Following could occur:

1. Borrower A has a loan connected to Coordinator A. Borrower B has a loan connected to Coordinator B. The lender is in both cases the same.
2. Borrower A and the lender renegotiate the lending terms.
3. Borrower B replays the signature while the signature is not expired yet.
4. The lender has renegotiated two positions instead of only one.

This attack works as long as the data provided to renegotiation functions is the same.

Code corrected:

Now, the contract address is signed. Thus, the signature can only be used on the valid contract.

6.5 SmartNFTs May Not Be Composable With Other Protocols

Design Medium Version 1 Code Corrected

When a loan is accepted, two SmartNFT tokens are issued: A promissory note NFT to the lender, and an obligation note NFT to the borrower. The NFT collateral is stored in the NFTfi loan contract until either the borrower repays the loan, or the loan is liquidated. However, when either of these events happen, the SmartNFT tokens are transparently destroyed, and the addresses owning the respective NFTs receive the collateral and payback. That makes the SmartNFTS untraceable by smart contracts. That could be

hazardous since the documentation specifies that a possible use-case of these NFTS could be trading them (e.g. selling the loan).

Assume the following scenario:

1. Lender and borrower agree on a loan, create it, and receive the SmartNFTs.
2. As time passes, the lender decides to sell the promissory note on a platform as a fixed-income debt-bearing asset. The promissory note is deposited into a smart contract.
3. Now, the borrower pays back the loan. Both SmartNFTs are burned. The collateral is transferred to the borrower. The payback is transferred to the NFT trade platform.

Ultimately, the auction of the promissory note cannot be ended. Hence, funds could get locked in the other contract while the lender does not receive anything.

Similarly, if the SmartNFTs are whitelisted in the PermittedList, funds could get lost in the NFTfi system since SmartNFTs could disappear at any time while a loan contract owning them would be clueless. Also, in such a way ImmutableBundles could lose funds.

To conclude, the immediate burning of SmartNFTs could be hazardous for NFTfi and other platforms as they could disappear at any point in time.

Code corrected:

Now, only whitelisted contracts or EOAs can hold SmartNFTs. Thus, governance must ensure that whitelisted contracts hold SmartNFTs that handle the scenarios above correctly.

6.6 Undeployable SmartNFTs

Correctness Medium Version 1 Code Corrected

SmartNFTs are used for the promissory note and obligation receipt. This contract inherits from OpenZeppelin's access control contract. The deployment of the contract may fail.

```
_setupRole(DEFAULT_ADMIN_ROLE, _admin);
grantRole(LOAN_COORDINATOR_ROLE, _loanCoordinator);
```

It sets `_admin` as the default administrator for all roles. If `_admin` is not `msg.sender`, then `grantRole` will fail.

Code corrected:

`_setupRole()` is now used instead of `grantRole` in the constructor.

6.7 Anyone Can Liquidate

Design Low Version 1 Code Corrected

The renegotiation feature allows to renegotiate even if the loan has expired. However, anyone can liquidate a loan. Thus, it could be possible that the result is not what the users desired. Moreover, fees that could have been earned will not be received.

Code corrected:

Now, only the lender can liquidate.

6.8 Double Getters

Design Low Version 1 Code Corrected

For each public variable, a getter is automatically generated. However, several contracts implement additional getter functions for public variables which leads to more code and, hence, higher deployment cost.

Some examples of double getters are:

- `partnerRevenueShare` and `getPartnerPermit` in PermittedPartners.sol
- `nftPermits` and `getNFTPermit` in PermittedNFTs.sol
- `erc20Permits` and `getERC20Permit` in PermittedNFTs.sol

Similar examples can be also found in other contracts such as DirectLoanCoordinator, NftfiHub and others. Removing double getters may reduce deployment cost.

Code corrected:

The double getters have been removed by setting the public variables to `private`.

6.9 Event Issues

Design Low Version 1 Code Corrected

Many events are emitted in the system helping users and front-ends. However, some event could be `indexed` to improve the experience. For example:

- The permitted list contracts could index the address of the permitted contract.
- Registry and loan contracts could have also indexed events

Furthermore, some important state changes do not emit events (e.g. `updateMaximumLoanDuration` or `updateMaximumNumberOfActiveLoans`). Note that also the renegotiation lacks events.

Emitting more events and indexing some of their parameters could improve the user-experience.

Code corrected:

The events are now indexed and more events are emitted.

6.10 Gas Inefficiencies

Design Low Version 1 Code Corrected

Structs are passed to the loan functions as arguments. These structs are passed compactly since they use for example `uint32`. However, some state variables could follow this principle. For example, `adminFeeInBasisPoints` will never be greater than 10000 but is a `uint256`. The structs store this as a `uint32`. However, a smaller data type could also be sufficient. Similar gas optimizations could be made.

Furthermore, since the hub should not change, it could be made `immutable` in all contracts. For example, DirectLoanCoordinator stores it as an immutable while DirectLoanBase does not. Similar gas savings could be achieved.

Also, some state variable may be redundant. For example, the loan status stored in the loan coordinator. It is only used for checking something when burning the receipt NFTs. However, burning requires the NFT owner to not be zero. Thus, the burn requirements are equivalent to the status checks.

Several retrieved values from storage and from other contracts could be cached in memory instead of reading it multiple times. For example:

- The NFT wrapper is retrieved in `loanSanityChecks` and when setting up the loan terms.
- `loanIdToLoan[id]` is read from storage into memory in `payBackChecks` and then in `payBackLoan`.

Further redundant storage reads can be found.

Moreover, `DirectLoanFixed._payoffAndFee` is computed as follows:

```
uint256 interestDue = _loanTerms.maximumRepaymentAmount - _loanTerms.loanPrincipalAmount;
uint256 adminFee = _computeAdminFee(interestDue, uint256(_loanTerms.loanAdminFeeInBasisPoints));
uint256 payoffAmount = ((_loanTerms.loanPrincipalAmount) + interestDue) - adminFee;
```

However, the addition could be removed since its result should be the maximum repayment amount.

Overall, gas consumption could be reduced by storing data more compactly, by reducing the number of storage reads and writes, and by removing redundant calculations.

Code corrected:

Gas consumption has been reduced.

6.11 Gas Inefficiencies in SmartNFTs `supportsInterface()`

Design | Low | Version 1 | Code Corrected

NFTs must implement EIP-165's proposed method `supportsInterface()`. SmartNFT implement this method. Gas could be saved there by calling only the `super` method which would, in this case, evaluate all the implementations of the parent classes and cover all implemented interfaces.

Moreover, deployment cost could be reduced by reducing the code size by using the methods and modifiers inherited from AccessControl. Thus, duplicated code could be removed.

Code corrected:

The gas consumption of the code has been optimized.

6.12 Maximum Number of Loans May Be Violated

Correctness | Low | Version 1 | Code Corrected

The administrator is allowed to specify a maximum number of loans allowed. The following invariant should always hold:

```
totalActiveLoans <= maximumNumberOfActiveLoans
```

However, that could be violated. Assume that these are equal. Then, the administrator calls `updateMaximumNumberOfActiveLoan` to reduce the maximum number of active loans. Ultimately, the invariant could be violated.

Code corrected:

An additional check was added to ensure that the invariant does not break.

6.13 Maximum Repayment Amount

Correctness Low Version 1 Code Corrected

The maximum repayment amount is specified by the lender. For both existing loan types this is value is relevant for accepting an offer while unused for accepting listings. The maximum repayment amount is calculated as the sum of the principal loan amount and the interest rate. However, that could be irritating for lenders as they could expect the maximum repayment amount specified by them to be used as the maximum.

Furthermore, in the pro-rated contract, renegotiation could lead to a scenario where the interest could grow even after time has elapsed since the interest rate is not modified.

Code corrected:

The maximum repayment amount specified by the lender is now always used. Also, the interest rate is now updated for the pro-rated loan.

6.14 Not Using `safeTransfer` for ERC-20 Transfers

Design Low Version 1 Code Corrected

Since not all ERC-20 tokens adhere to the standard, it is recommended to use `safeTransferFrom` such that interactions with a broader range of tokens are possible. However, the transfer of the renegotiation fee does not use the safe operation.

Code corrected:

`safeTransferFrom` is now used.

6.15 Renegotiation on Wrong Contract Possible

Design Low Version 1 Code Corrected

Renegotiation is a feature that allows the lender to give the borrower an alternative offer after the loan has been created. However, it could be possible to renegotiate a loan on the wrong contract.

This becomes possible if the maximum loan duration is greater than the current block timestamp and no renegotiation fee is charged.

1. Lender and borrower agree on a direct fixed loan.
2. Lender signs the renegotiation with a high new loan duration for the pro-rated contract.
3. The borrower calls renegotiate on the pro-rated loan contract.
4. The correct SmartNFT ID is fetched from the shared coordinator while the loan data is empty as it is stored per lending contract.
5. Thus, if the maximum loan duration and the new loan duration are sufficiently high and the renegotiation fee is 0 (no ERC-20 transfer occurs), all checks pass.

However, as the NFT wrappers are not initialized, this leads to unnecessary state modifications while funds cannot be transferred.

Code corrected:

Now, the loan contract is compared to the stored contract in the loan coordinator disallowing such wrong renegotiations.

6.16 Repetitive Validation on Batch Child Transfer

Design | Low | Version 1 | Code Corrected

`safeBatchTransferChild()` allows children of a token to be batch transferred. `msg.sender` is validated in each loop iteration to be the root owner of `tokenId`. However, since only the children of one token id can be batch transferred at once, it is sufficient to validate only once. Ultimately, storage reads and, hence, gas consumption could be reduced.

Code corrected:

The method has been optimized.

6.17 Specification Mismatch

Correctness | Low | Version 1 | Specification Changed

The code has several occurrences of specification mismatch. Some examples are:

- `DirectLoanProRated._setupLoanTermsListing` documents that it is a fixed loan.
- `ERC998TopDown.childExists` specifies that it returns `true` if a child exists. However, in the extended classes this will return `false` for ERC-1155 tokens.
- `ERC998TopDown.ownerOfChild` specifies that parameter `tokenId` while it has only parameter `childTokenId`.

Specification changed:

The specification has been updated.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Fee Avoidance

Note Version 1

The liquidation allows the administration fees to be avoided as follows:

1. The borrower transfers his receipt to a contract. As long as the lender has transferred his receipt to the contract, the borrower can withdraw his receipt.
2. The lender signs a renegotiation and approves the cheating contract.
3. The lender calls the cheating contract method that takes the renegotiation and the renegotiation parameters as arguments (and checks whether the parameters are fair).
4. The contract, having the obligatory note, calls renegotiate.
5. The contract pulls the promissory note from the lender.
6. The loan gets liquidated and the contract holds the NFT collateral.
7. The cheating contract implements a payback function that is cheaper for the borrower and more profitable for the lender (splitting the admin fee). Moreover, as a safeguard for the lender it implements a liquidation function.

Ultimately, no fees are distributed to the administration while the lender and borrower could profit.

This behaviour cannot occur anymore. Since only EOA addresses could hold a SmartNFT in such a case, the lender would need to trust the borrower.

7.2 Front-running Offers

Note Version 1

Alice may receive an Offer of Bob for an ERC-1155 token. Charlie could call `acceptOffer()` with Bob's signature which would initiate a loan between Bob and Charlie for the same ERC-1155 token. However, Bob's intend could have been to only allow Alice to take a loan from him. From the discussions with NFTfi, it was clarified that the ERC-1155 tokens to be supported are the ones that have at most one token per ID.

Hence, governance needs to be careful when whitelisting ERC-1155 contracts.

7.3 Outdated Compiler Version

Note Version 1

The solc version is fixed in the hardhat configuration to version `0.8.4`. At the time of writing the most recent Solidity release is version `0.8.7`.

7.4 Possible Inconsistencies After Registry Changes

Note Version 1

Many values are stored such that the loan can be resolved, no matter the changes made to the system (e.g. whitelisting ERC-20 tokens). However, the loan registry is global for the whole system. That could introduce several issues:

- Assume contract A is stored in the loan registry for loans of type B and Loans are still active. Now, administration changes the contract for loan type of B to contract C. That could lock the funds in contract A and make the loans unresolvable or introduce other issues related to that.
- The loan coordinator could change in the hub. Thus, loans could become all invalid since changing the new loan coordinator could also change the smart NFT token contract address.
- Loans could become unresolvable if the loan coordinator loses access to a Smart NFT contract.

These and similar issues could occur.

7.5 Supported Tokens

Note Version 1

The protocol supports ERC-20 tokens as lending capital. However, whitelisting for example ERC-777 tokens (backward-compatible with EIP-20) may lead to unwanted behaviour. For example, paybacks of loan could be blocked by reverting on token reception.

Also, borrowers may receive less than expected if the ERC-20 tokens collect transfer fees while the paybacks could fail.

Furthermore, some NFTs could be added that could be burnable externally or have other unexpected non-standard behaviour.

In general, governance has to be careful with whitelisting tokens.