vacuumlabs

FluidTokens Peer-to-Peer Lending (EVM)

Audit Report v1

September 27, 2023

Contents

Revision table

Report version	Report name	Date	Report URL
1.0	Main audit	2023-09-27	Full report link

1 Executive summary

THIS REPORT DOES NOT PROVIDE ANY WARRANTY OF QUALITY OR SECURITY OF THE AUDITED CODE and should be understood as a best efforts opinion of Vacuumlabs produced upon reviewing the materials provided to Vacuumlabs. Vacuumlabs can only comment on the issues it discovers and Vacuumlabs does not guarantee discovering all the relevant issues. Vacuumlabs also disclaims all warranties or guarantees in relation to the report to the maximum extent permitted by the applicable law. This report is also subject to the full disclaimer in the appendix of this document, which you should read before reading the report.

Project overview

The project offers a *peer-to-peer decentralized lending with an NFT collateral*. The smart contract is intended to be used on Milkomeda (Cardano EVM Sidechain) first, but the code could work on every EVM chain. The contract is the first step of FluidTokens to showcase the EVM capabilities and to increase their community. A person interested in borrowing some assets (a borrower) can create a loan request that contains all the loan information, including an amount and an asset to be lent, an interest amount, a duration of the loan, and a number of installments in which the loan will be paid back. They back this request with the collateral that is locked in the contract. *The collateral is a single NFT.* The borrower can cancel the loan request before it is accepted.

Anyone can accept this loan request (a lender) by sending an appropriate amount of the specified loan asset to the borrower. The borrower is then obliged to pay back the loan amount with the interest according to the agreed-upon terms. The total loan and interest amounts are split evenly among the whole loan duration and need to be repaid regularly – the first installment is due in the first portion of the total loan duration, etc.

If any single loan repayment is not paid on time, the lender can claim the collateral – the underlying NFT. There are no other liquidation options, e.g. there's no liquidation option because of a drop in the value of the collateral backing the loan.

The access to *a loan is managed by bond tokens* – one is minted for a borrower and one is minted for a lender, both when the lender is accepting the loan. The borrower's bond token can be used to pay an installment. The lender's bond token can be used to claim a repayment or to claim the collateral in case a repayment is not repaid on time.

As only these tokens control access to the mentioned functions, they can be sold or moved to other addresses. *The responsibilities, rights, as well as entitlements of the* *respective party are transferred alongside the bond token ownership.*

Audit overview

We started the audit at commit `b806d2ffdbc216c14e0695b8f776745729b77d9e` and it lasted from 28 July 2023 to 27 September 2023. Because of multiple needed iterations and because of the fact that this was our first EVM audit, the audit lasted longer than a typical audit of a similar size would. We interacted mostly on Discord and gave feedback in GitHub pull requests. The team fixed all issues to our satisfaction.

The scope of the audit was limited to the smart contracts only. We did not audit any OpenZeppelin or other libraries that were used in the code. We did not review nor see any tests as part of this audit, and no tests were included in the repository.

As a suggestion for further enhancing the codebase, we recommend integrating tests into the repository and incorporating them into the regular development workflow. We believe that such a step would proactively identify and resolve various issues we encountered in the pull requests addressing the findings outlined in this report.

We performed a design review along with a deep manual audit of the code, ran static analysis tools and reported findings along with remediation suggestions to the team in a continuous fashion, allowing the time for a proper remediation that we reviewed afterwards. We also supplied proof of concepts demonstrating the vulnerabilities on multiple occasions. See more about our methodology in Methodology.

The commit `95650b663baddbbf07d8fcf489b8a8e1473f7a32` represents the final version of the code. The status of any issue in this report reflects its status at that commit. You can see all the files audited and their hashes in Audited files. The smart contract language used is Solidity and the contracts are intended to run on Milkomeda – a Cardano EVM sidechain.

Summary of findings

During the audit, we found and reported: 3 critical, 5 major, 4 medium, 7 minor, and 8 informational findings. All findings were fully resolved.

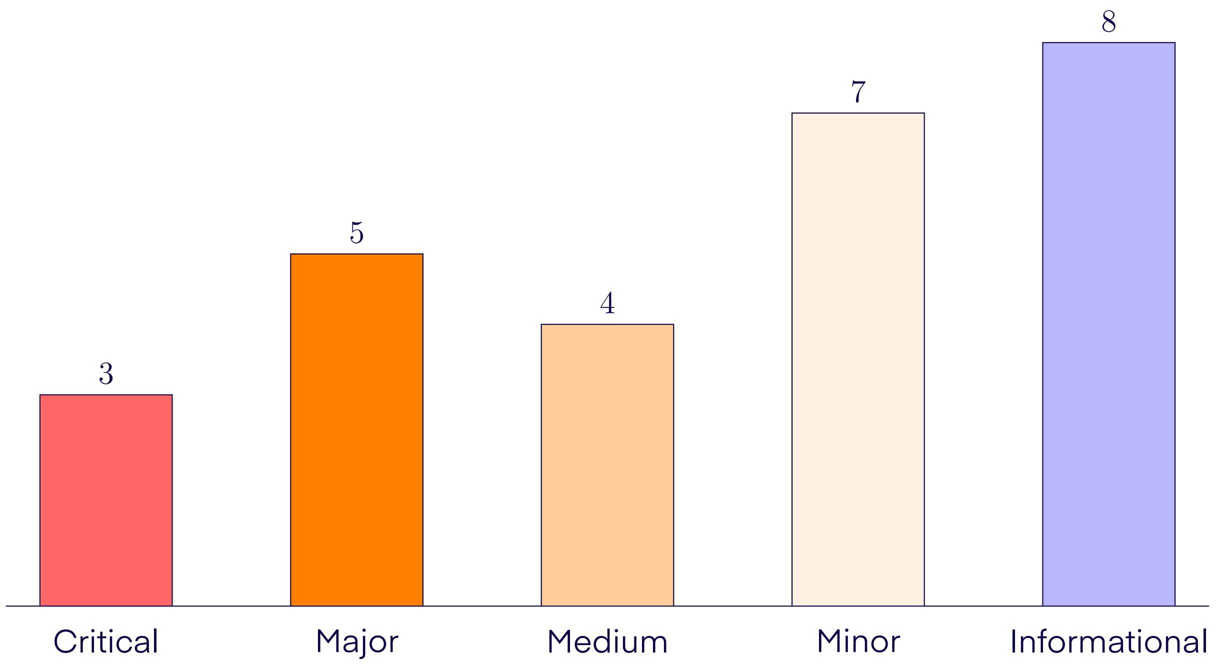
Some issues were caused by the overall design and a lack of documentation. This affected mostly the bond tokens – their handling and the transfer of responsibilities. We have helped the Fluid Tokens' team clarify and simplify the design of the smart contract.

Another set of vulnerabilities was caused by various possible reentrancies. Although the design contained some prevention measures that were implemented, they changed during the rework and were not sufficient anymore. For this, a more robust reentrancy guard was implemented using an OpenZeppelin library that prevents the reentrancy altogether.

There were also issues with the token transfers, making it possible for them to become either stolen or frozen in the contract.

The code went through several changes. Some of those changes introduced new vulnerabilities and bugs, even of critical severity, that are not present in this report. These issues were small in scope and would be categorized as Code issues. They were spotted during the pull request review process and were quickly fixed so we haven't created separate issue findings for them. The fact is sometimes noted in the Resolution section of the relevant findings, though.

2 Severity overview



Findings

ID	TITLE	SEVERITY	STATUS
FTEA-001	Contract tokens can be stolen due to incorrect usage of `approve` and `transfer`	CRITICAL	RESOLVED
FTEA-002	Borrower needs twice the borrowed balance to repay a native token loan	CRITICAL	RESOLVED
FTEA-003	Borrower can withdraw collateral without repaying the loan	CRITICAL	RESOLVED
FTEA-101	Withdrawing the last repayment locks other unclaimed repayments	MAJOR	RESOLVED
FTEA-102	Unclaimed repayments are locked after the collateral is claimed	MAJOR	RESOLVED

Continued on next page

ID	TITLE	SEVERITY	STATUS
FTEA-103	Usage of `transferFrom` method for ERC-20 tokens	MAJOR	RESOLVED
FTEA-104	Missing incentives for repaying in installments	MAJOR	RESOLVED
FTEA-105	Loan and collateral are transferred to the borrower instead of the bond owner	MAJOR	RESOLVED
FTEA-201	Loan request expiration is not enforced	MEDIUM	RESOLVED
FTEA-202	Lender needs twice the required balance to lend to a native token loan	MEDIUM	RESOLVED
FTEA-203	Possible to flash loan any contract's token for the transaction	MEDIUM	RESOLVED
FTEA-204	Events can be emitted with wrong data due to reentrancy	MEDIUM	RESOLVED
FTEA-301	Anyone can setup the contracts	MINOR	RESOLVED
FTEA-302	Incorrect loan and interest calculation in the last installment	MINOR	RESOLVED
FTEA-303	Duplicated storage of data	MINOR	RESOLVED
FTEA-304	NFTs can be locked in the contract	MINOR	RESOLVED
FTEA-305	Repayment structure is not necessary	MINOR	RESOLVED
FTEA-306	Excessive repayment is recorded as a single installment repaid	MINOR	RESOLVED
FTEA-307	Dependencies are not committed into the repository	MINOR	RESOLVED
FTEA-401	Floating pragma	INFORMATIONAL	RESOLVED
FTEA-402	Usage of unnamed constants	INFORMATIONAL	RESOLVED

Continued on next page

ID	TITLE	SEVERITY	STATUS
FTEA-403	Usage of public functions where external can be used	INFORMATIONAL	RESOLVED
FTEA-404	Unnecessary check of collateral token’s owner and approval	INFORMATIONAL	RESOLVED
FTEA-405	No `tokenData` clean-up after bond’s burn	INFORMATIONAL	RESOLVED
FTEA-406	Bonds’ operator can not act on behalf of the owner	INFORMATIONAL	RESOLVED
FTEA-407	Use `call` instead of `transfer` to move ETH	INFORMATIONAL	RESOLVED
FTEA-408	Code style issues	INFORMATIONAL	RESOLVED

FTEA-001 Contract tokens can be stolen due to incorrect usage of `approve` and `transfer`

Category	Vulnerable commit	Severity	Status
Logical Issue	`b806d2ffdb`	CRITICAL	RESOLVED

Description

Assuming the interest tokens or the loan tokens are ERC-20 tokens, the contract both approves and transfers the tokens to the lender in the `withdrawRepayment` function. However, the `transfer` function call in the ERC-20 standard does not reset the allowance from the `approve` call – as opposed to calling the `transferFrom` function which correctly updates the allowance.

This means that the lenders can get the tokens twice – the first time, they are sent to them directly using the `transfer` function; then they can withdraw the same amount themselves again because it was approved by the `approve` function.

This allows an attacker to drain all the liquidity from the contract. First, they create a loan for some amount of tokens. Then, they lend the tokens to themselves, instantly repaying them and withdrawing twice the amount. They repeat the process until no ERC-20 tokens are left in the contract.

Recommendation

The `approve` function calls are not necessary in the code and it is possible to simply remove them. Calling the `transfer` function is enough.

Resolution

The issue was fixed according to our recommendation in the pull request number 1.

FTEA-002 Borrower needs twice the borrowed balance to repay a native token loan

Category	Vulnerable commit	Severity	Status
Logical Issue	`b806d2ffdb`	CRITICAL	RESOLVED

Description

Let's assume that either the loan or the interest token is native (the token address is set to the zero address). When repaying the loan, the code checks that the balance of the message sender is greater than the amount he is paying in the `repayLoan` function. As the `msg.value` is excluded from the `msg.sender.balance`, this check is redundant. What's more, it prevents a repayment if the balance of the borrower is low. As the function checks both the funds sent in the transaction and the outstanding balance of the borrower, the borrower needs to keep in his wallet as much tokens as he is repaying for it to succeed.

As an example, let us consider a borrower holding 0 Eth that is borrowing 50 against his NFT. He would need to have 100 Eth to be able to repay it all in one installment. In particular, he needs to satisfy both the following checks: `msg.sender.balance >= installmentLoanAmnt` and `availableTxValue >= installmentLoanAmnt`.

This vulnerability can lead to liquidations of users who could repay their loans fully, but do not own enough additional tokens. We consider the vulnerability to be critical, because fair liquidations are the basis of trust in this type of contracts.

Recommendation

The check of `msg.sender.balance` is unnecessary in this situation and we recommend removing the checks using it.

Resolution

The issue was fixed according to our recommendation in the pull request number 1.

FTEA-003 Borrower can withdraw collateral without repaying the loan

Category	Vulnerable commit	Severity	Status
Logical Issue	`26864b1ffb`	CRITICAL	RESOLVED

Description

Let's assume an attacker that creates a valid loan request with an NFT collateral. They will set a `borrower` address to their own malicious smart contract that implements the `onERC721Received` function. The `onERC721Received` can then perform a reentrancy attack.

When the loan is accepted, the `lend` function mints a bond token and sends it to the `borrower` address, triggering the `onERC721Received`. Let's make that function call the `cancelLoan` function. All requirements for the cancellation are fulfilled, so the borrower withdraws the collateral. However, the `lend` function execution is also successful and the borrower receives the loan amount as well. The loan is not backed by any collateral and both the loan and the collateral are claimed by the attacker.

Recommendation

This reentrancy attack can be prevented by following the checks-effects-interactions design pattern. If the loan status is set to `ACTIVE` before the minting of bond tokens, the cancellation won't be possible anymore.

Resolution

The issue was fixed in the pull request number 8 by implementing reentrancy guards as recommended in FTEA-203.

FTEA-101 Withdrawing the last repayment locks other unclaimed repayments

Category	Vulnerable commit	Severity	Status
Design Issue	`b806d2ffdb`	MAJOR	RESOLVED

Description

When the last repayment is withdrawn, the bond token of the lender is burned in the `withdrawRepayment` function. If the repayments are not withdrawn in order, the rest of the repayments is locked in the contract forever. Neither the lender nor the project can access them.

The frequency of the scenario happening depends much on the UI. However, as multiple repayments need to be withdrawn in multiple function calls, we consider it likely.

Recommendation

We recommend tracking the number of withdrawn repayments for each loan – either in the contract or in the bonds token itself. Burn the token only after all the repayments are withdrawn.

Resolution

The issue was fixed in the pull request number 2 indirectly by addressing the issue FTEA-305 first and making sure that any repayment withdrawal claims all the outstanding amounts.

FTEA-102 Unclaimed repayments are locked after the collateral is claimed

Category	Vulnerable commit	Severity	Status
Design Issue	`b806d2ffdb`	MAJOR	RESOLVED

Description

When a borrower can not repay an installment on time, the lender can claim the collateral. However, claiming the collateral burns the lender's bond token. Any repayments that have been made by the borrower and have not been claimed by the lender are rendered inaccessible and are locked inside the contract.

This handling is unintuitive for the users. Similar to the previous issue FTEA-101, the frequency of the issue occuring depends on the used UI. However, a straightforward UI is vulnerable and so we deem it of major severity. The impact is that the unclaimed repayments would accumulate inside the contract with no way to retrieve them.

Recommendation

Only burn the lender's bond token when there are no more repayments in the contract.

Resolution

The issue was fixed according to our recommendation in the pull request number 4.

FTEA-103 Usage of `transferFrom` method for ERC-20 tokens

Category	Vulnerable commit	Severity	Status
Logical Issue	`b806d2ffdb`	MAJOR	RESOLVED

Description

All ERC-20 tokens should implement the `transferFrom` method that returns a boolean as a result. However, not all ERC-20 tokens are compliant with the standard, including some of the well known tokens. Most notably, the Tether (USDT) token uses a `transferFrom` method that does not return any value[1].

This can cause a critical failure in the smart contract. Let's assume that the loan is created and the `interestToken` is set to Tether or a similar non-compliant ERC-20 token. If the loan is accepted, the installments are impossible to be paid as the `require` statement for the `transferFrom` function call of the token is not fulfilled due to the missing return value. Such loans will be liquidated every time even if the borrowers have the means to pay them back.

Recommendation

When working with ERC-20 tokens, one needs to assume that not all token implementations are compliant with the standard. For this use case, the `SafeERC20` library provided by OpenZeppelin can be used. The library handles such cases and returns corrected values compliant with the standard.

Resolution

The issue was fixed according to our recommendation in the pull request number 2.

[1]https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7#code

FTEA-104 Missing incentives for repaying in installments

Category	Vulnerable commit	Severity	Status
Design Issue	`b806d2ffdb`	MAJOR	RESOLVED

Description

The implementation supports repaying a loan in several installments whose number is agreed upon in the loan request. However, there is no incentive for anyone to use this option. The smart contract only checks that all the installments are repaid before the end of the loan period. Similarly, a liquidation is only possible after the loan period is over and one or more installments are still not repaid.

Let's say that a borrower can not pay the last installment. In such a case she is liquidated and she also loses all her previously paid installments. Therefore, the incentive for borrowers is to pay all the installments at the end of the loan period. That means that the lender can not expect any payments to be made before the end of the period. That makes the option of setting up several installments in the contract unnecessary and the code related to it nonessential.

Recommendation

Depending on the business logic, the code could either be simplified by removing the option to pay in installments, or it should introduce an incentive to adhere to the repayment schedule – e.g. by adding a liquidation option as soon as any installment is not repaid in time.

Resolution

The code introducing an incentive to adhere to the repayment schedule was introduced in the pull request number 1. A borrower may be liquidated if she is late with repaying any installment.

FTEA-105 Loan and collateral are transferred to the borrower instead of the bond owner

Category	Vulnerable commit	Severity	Status
Design Issue	`6b5f7a7ea1`	MAJOR	RESOLVED

Description

The bond tokens are intended to be traded with all their responsibilities, rights and entitlements. However, the loan is transferred directly to the initial creator of the loan request. The collateral is also returned directly to the original borrower.

This makes the option of transferring the borrower's bond token impractical as the new owner would not be able to retrieve the collateral locked in the loan even if he repays the loan with interest. As a result, there is no incentive for him to do it.

Recommendation

Instead of the original borrower, the recipient of the loan payment and, more importantly, that of the collateral should be the owner of the borrower's bond token or an entity entitled to act on his behalf – the `msg.sender` in the case of the last `repayLoan` function call. In fact, the `borrower` parameter in the `LoanRequestData` may become obsolete and could be removed.

Resolution

The loan is still paid to the initial borrower, but the code enabling a claim of the collateral now transfers it to the current owner of the bond token. The changes were implemented in pull requests number 6. The change, however, contained a bug where the borrower's bond token was burned before the collateral was transferred to its owner, thus causing the borrower to never get back the collateral – even after the loan repayment. This bug was found during the pull request review process and correctly fixed in the pull request number 7.

FTEA-201 Loan request expiration is not enforced

Category	Vulnerable commit	Severity	Status
Logical Issue	`b806d2ffdb`	MEDIUM	RESOLVED

Description

The `requestExpiration` configuration is not enforced in the code. Therefore, calling the `lend` function on an expired loan leads to a succesful loan creation. The expected behavior is that the only interaction allowed with an expired loan should be its cancellation by its owner. Borrowers rely on the contract ensuring that the expiration date is enforced. They may not want to borrow after a certain time.

Recommendation

Check the `requestExpiration` in the `lend` function against the current block timestamp.

Resolution

The issue was addressed in the pull request number 1. The fix, however, did not address the issue correctly. It checked that the `requestExpiration` was equal to the current block timestamp during lending which made it impossible for lenders to lend to any loan request. A potential lender would need to lend exactly in the moment of the loan expiration which is near impossible.

We noticed this bug during the pull request review process and it was fixed in the pull request number 3.

FTEA-202 Lender needs twice the required balance to lend to a native token loan

Category	Vulnerable commit	Severity	Status
Logical Issue	`b806d2ffdb`	MEDIUM	RESOLVED

Description

When lending to a loan with the native loan token (the token address is set to the zero address), the `lend` function checks that the balance of the message sender is greater than or equal to the amount that is being lent. This check is redundant and prevents him from lending in certain conditions. That is because the `msg.value` is already excluded from the `msg.sender.balance`. As the function checks both of them, the lender needs to keep in his wallet as much tokens as he is lending for it to succeed.

This vulnerability has similar root cause as FTEA-002. However, as this vulnerability does not cause unfair liquidations we set lower severity to it.

Recommendation

The check of the `msg.sender.balance` is unnecessary in this situation and we recommend removing it.

Resolution

The issue was fixed according to our recommendation in the pull request number 1.

FTEA-203 Possible to flash loan any contract's token for the transaction

Category	Vulnerable commit	Severity	Status
Logical Issue	`26864b1ffb`	MEDIUM	RESOLVED

Description

The current implementation of the protocol allows anyone to loan any amount of any token that is present in the contract for the duration of the transaction – also called a flash loan. The protocol neither aims to provide such a functionality nor receives any fees for it.

The attack is performed by a borrower creating a loan request specifying a big interest amount of a wanted token – the amount and the token he wants to lend – and a malicious collateral token or a loan token address. He then himself fulfills the loan request. Note that the loan amount is not important.

The flash loan happens in the `repayLoan` function after both the `totalRepaidLoan` and the `totalRepaidInterest` are incremented but the tokens are not yet transferred. It is possible for him to reenter into the `withdrawRepayment` function from either his maliciously set collateral asset or the loan asset. By doing so, he is able to withdraw any amount of the interest token before it is repaid. Although the lent amount needs to be returned, this exploit can be used to manipulate the market or to gain an unfair advantage.

Recommendation

To prevent this exploit, we recommend implementing a non-reentrant mutex across all main state modifying functions. This can be achieved by using OpenZeppelin's `ReentrancyGuard` contract. This will also help to prevent other reentrancy attacks related to manipulating event values. You can find more information about this in the OpenZeppelin documentation[2].

Resolution

The issue was fixed according to our recommendation in the pull request number 8.

[2]https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard

FTEA-204 Events can be emitted with wrong data due to reentrancy

Category	Vulnerable commit	Severity	Status
Logical Issue	`26864b1ffb`	MEDIUM	RESOLVED

Description

In several functions, it is possible to create an event with wrong data due to reentrancy. It does not pose any security risk to the on-chain protocol but results in wrong events being emitted which are likely heavily depended upon in the other application components.

For example, suppose that during the `repayLoan` call, another `repayLoan` is called as a result of a reentrance from a token transfer. The loan variable `currentInstallment` will be raised two times and this new value will be used for both the `LoanRepaid` events. Therefore, two events with the same `currentInstallment` will be emitted.

A similar attack is feasible in the `createLoan` function.

Although this behavior can not be used to steal funds or otherwise compromise the protocol on-chain, it can affect any off-chain application that relies on these events.

Recommendation

This issue could be fixed in various ways – either by strictly following the checks-effects-interactions design pattern or by using only local variables to emit the events. However, we recommend using OpenZeppelin's `ReentrancyGuard` contract that can protect functions from reentrancies as was already recommended in the previous finding FTEA-203.

Resolution

The issue was fixed in the pull request number 8 by implementing reentrancy guards as recommended in FTEA-203.

FTEA-301 Anyone can setup the contracts

Category	Vulnerable commit	Severity	Status
Logical Issue	`b806d2ffdb`	MINOR	RESOLVED

Description

The `setupPlatform` function is public, therefore available for anyone to call. As a result, anyone can front-run the call and use their own malicious `bondsAddress`. A contract initialized that way would need to be redeployed. However, if noticed, it could all be re-done before users interacted with the contract.

Recommendation

We recommend one of the two approaches:

- You could write a new *deployer* smart contract that would properly and, most importantly, atomically initialize both the `bonds.sol` and the `ft_p2p.sol`.
- You could introduce an owner variable to the `ft_p2p.sol` contract and allow only the owner to call the `setupPlatform` function.

Resolution

The issue was fixed by introducing a constant address that belongs to Fluid Tokens and checking it to be equal to the caller of the `setupPlatform` function across the pull requests number 1 and 2.

FTEA-302 Incorrect loan and interest calculation in the last installment

Category	Vulnerable commit	Severity	Status
Code Issue	`b806d2ffdb`	MINOR	RESOLVED

Description

The functions `getLoanAmountForInstallment` and `getInterestAmountForInstallment` calculate the loan and interest amounts that should be paid in an installment. The first installments simply divide the overall value to be paid by the number of installments. That creates rounding errors that are covered in the last installment. However, the final calculation is incorrect due to missing parentheses.

Due to the flooring nature of the integer division in Solidity, we have the following:

$$\lfloor a \cdot b/c \rfloor \geq a \cdot \lfloor b/c \rfloor$$

The overall difference between the two expressions is at most a – the number of installments in our case. Therefore, less funds need to be paid in total creating a slight loss for the lender.

Recommendation

Fix the formulas for the calculation of the last installment and interest amounts to correctly reflect the already repaid value.

Resolution

The issue was fixed according to our recommendation in the pull request number 2.

FTEA-303 Duplicated storage of data

Category	Vulnerable commit	Severity	Status
Optimization	`b806d2ffdb`	MINOR	RESOLVED

Description

The same information is often stored in multiple places of the code. For example, the bonds token keeps track of all the loan's parameters, even though the data is not used anywhere else in the smart contract. The impact of this can be twofold:

- Optimization problems – by keeping the same information in multiple places, the gas costs increase.
- Possible bugs – when multiple components track the same data, there is an increased possibility of introducing bugs if these components become desynchronized.

Recommendation

We recommend removing all unused variables from structures to simplify the code and reduce gas costs. We also suggest identifying a single source of truth for each piece of information and storing it solely there.

Resolution

The issue was fixed according to our recommendation in the pull request number 2.

FTEA-304 NFTs can be locked in the contract

Category	Vulnerable commit	Severity	Status
Logical Issue	`b806d2ffdb`	MINOR	RESOLVED

Description

The contract implements the `IERC721Receiver` interface. According to the documentation, the purpose of this interface is to prevent NFTs from becoming forever locked in contracts. However, the `onERC721Received` function does not check that the token is sent to the contract during the `createLoan` call. Therefore, any NFTs that are sent directly to the contract are locked forever and the interface is not implemented correctly.

Recommendation

As the contract can only receive NFTs in one specific use case (during the `createLoan` call), the `onERC721Received` function should revert when NFTs are sent to the contract in any other way. You can check the caller of the `safeTransferFrom` by looking at the `operator` parameter.

Resolution

The issue was fixed according to our recommendation across the pull requests number 2 and 4.

FTEA-305 Repayment structure is not necessary

Category	Vulnerable commit	Severity	Status
Design Issue	`b806d2ffdb`	MINOR	RESOLVED

Description

Currently, a borrower and a lender agree upon a fixed number of repayments. Then, all the installments need to be paid across separate transactions. What's more, all the repayments need to be claimed by the lender in separate transactions as well.

In addition to that, the `Repayment` structure unnecessarily duplicates some fields and thus wastes gas. It also requires counters to keep track of how many repayments were paid, how many were withdrawn; and thus complicates the code.

Recommendation

We recommend redesigning the application and removing the `Repayment` structure altogether. Instead, we suggest keeping track of the total repaid loan and interest amounts and the total withdrawn loan and interest amounts. You can still enforce fixed repayment amounts if you want to. An added benefit of the redesign is a support for multiple repayments/withdrawals in a single transaction.

Resolution

The repayment structure was changed according to our recommendation in the pull request number 2. In the new repayment structure, the value of the `totalWithdrawnLoan`, the amount the lender has withdrawn from the repayments, was not updated when the lender was withdrawing a repayment. This caused a bug where a lender could drain the whole contract by withdrawing the repayment multiple times. We noticed this bug during the pull request review process and it was fixed in the pull requests number 3 and 4.

FTEA-306 Excessive repayment is recorded as a single installment repaid

Category	Vulnerable commit	Severity	Status
Logical Issue	`b806d2ffdb`	MINOR	RESOLVED

Description

Currently, when multiple installments are sent to the `repayLoan` function, they are recorded as a single installment repaid. This issue concerns ETH repayments only and arises when the `msg.value` is greater than or equal to the installment loan plus interest amounts. This can be particularly problematic if users call the function directly, as they may unintentionally overpay without realizing that the excess funds are not properly accounted for. Furthermore, these excess funds become inaccessible and are locked within the contract.

Recommendation

We recommend either modifying the function to revert when the `msg.value` exceeds the expected installment amount or documenting and communicating this behavior clearly to inform users.

Alternatively, you could adjust the function to correctly record potentially multiple installments when the `msg.value` is greater than the installment loan plus interest amounts. The handling suggested in the issue FTEA-305 would enable accounting even for fractional repayments. This would prevent excess funds from being locked in the contract and ensure accurate tracking of repayments.

Resolution

The code reverts if the `msg.value` contains more than is necessary. The change was implemented in the pull request number 8.

FTEA-307 Dependencies are not committed into the repository

Category	Vulnerable commit	Severity	Status
Code Issue	`26864b1ffb`	MINOR	RESOLVED

Description

The contract's dependencies such as ERC721, SafeERC20 and others, are not committed into the repository. Some of those dependencies were provided upon request, but it is crucial to have them committed so that their specific version is fixed. This helps ensure that they are not unintentionally changed and guarantees that the same version that is tested is also the one deployed. It enhances the reliability and consistency of the codebase.

It is particularly important as an internal function of the ERC721 implementation is used in the `bonds` contract and there are differences in the naming of that function across different ERC721 implementations.

Recommendation

We recommend committing all dependencies into the repository.

Resolution

The issue was fixed according to our recommendation in the pull request number 8.

FTEA-401 Floating pragma

Category	Vulnerable commit	Severity	Status
Code Issue	`b806d2ffdb`	INFORMATIONAL	RESOLVED

Description

The files `bonds.sol` and `ft_p2p.sol` use a floating pragma `^0.8.7`. The code could therefore potentially be compiled by a different compiler than it was tested on.

Recommendation

We recommend fixing the pragma to a specific compiler version to prevent any unwanted inconsistencies between development and deployment.

Resolution

The issue was fixed according to our recommendation in the pull request number 1.

FTEA-402 Usage of unnamed constants

Category	Vulnerable commit	Severity	Status
Code Style	`b806d2ffdb`	INFORMATIONAL	RESOLVED

Description

The code uses unnamed integer constants to track the status of loans and repayments. Unnamed constants can lead to confusion and potential bugs as they lower the readability of the code.

Recommendation

We recommend changing the `status` variable in the Loan structure to an enum type, and to rename the boolean variable named status in the `Repayment` structure to `isWithdrawn` to better reflect the intended meaning.

Resolution

The issue was fixed according to our recommendation in the pull request number 2.

FTEA-403 Usage of public functions where external can be used

Category	Vulnerable commit	Severity	Status
Optimization	`b806d2ffdb`	INFORMATIONAL	RESOLVED

Description

All the functions in the code are public which cost more gas than external functions. Public functions should only be used when they are called both from the outside and from the contract itself.

Recommendation

We recommend changing the public functions that are not called internally to external.

Resolution

The issue was fixed according to our recommendation in the pull requests number 2 and 8.

FTEA-404 Unnecessary check of collateral token’s owner and approval

Category	Vulnerable commit	Severity	Status
Optimization	`b806d2ffdb`	INFORMATIONAL	RESOLVED

Description

In the `createLoan` function, one require statement checks that the collateral token is either owned by or approved to the `msg.sender`. However, the check is redundant as the function later calls `safeTransferFrom` to transfer the NFT from the `msg.sender` which can only succeed if the `msg.sender` owns the NFT. Also, the first part of the check `collateralToken.getApproved(_collateralTokenId) == msg.sender` can never pass, as the collateral token has to be approved to the smart contract and not to the `msg.sender`.

Recommendation

Remove the unnecessary check to save gas and simplify the code.

Resolution

The issue was fixed according to our recommendation in the pull request number 2.

FTEA-405 No `tokenData` clean-up after bond's burn

Category	Vulnerable commit	Severity	Status
Optimization	`b806d2ffdb`	INFORMATIONAL	RESOLVED

Description

The bonds' burn function successfully removes the NFT representation of the bond but leaves the associated `TokenData` in the contract's storage. This results in wasted storage and is inconsistent with the handling of the ownership data.

Recommendation

Add a `delete` statement to remove the `TokenData` associated with a burned bond. Additionally, consider emitting an event that logs the data before its deletion for off-chain tracking and analytics.

Resolution

The issue was fixed according to our recommendation in the pull request number 2.

FTEA-406 Bonds' operator can not act on behalf of the owner

Category	Vulnerable commit	Severity	Status
Logical Issue	b806d2ffdb	INFORMATIONAL	RESOLVED

Description

The bonds are ERC-721 tokens that represent transferable ownership and the corresponding duty to repay a loan or withdraw repayments. However, the protocol does not utilize ERC-721's built-in **operator** functionality even though it utilizes the `approve` functionality. Specifically, functions such as `lend`, `withdrawRepayment`, and `claimCollateral` validate the sender's permission by directly comparing him to the approved address or the owner of the token. This ignores the possibility that he may be an operator approved to manage all of the owner's tokens. The problematic code line (indentation is ours):

```
require(
  bonds.getApproved(loan.lenderBondTokenId) == msg.sender
    || bonds.ownerOf(loan.lenderBondTokenId) == msg.sender
);
```

This approach limits the flexibility and usability of these bonds, as it restricts the actions that can be performed by operators who are supposed to act on behalf of the owners.

Recommendation

To fully support ERC-721's approvals and to allow for more flexible management of bonds, the protocol should use a function similar to the `_isApprovedOrOwner` internal function provided by the OpenZeppelin's ERC-721 implementation attached. This function checks not only if the message sender is the current owner or an approved party for a specific `tokenId`, but also if he is an operator approved for all tokens owned by the current owner.

This change would allow operators to perform the essential functions on behalf of the owner, making the system more flexible and consistent with the ERC-721 standard.

Resolution

The issue was fixed according to our recommendation in the pull request number 3.

FTEA-407 Use `call` instead of `transfer` to move ETH

Category	Vulnerable commit	Severity	Status
Logical Issue	`6b5f7a7ea1`	INFORMATIONAL	RESOLVED

Description

The `transfer` function for sending ETH has a hard limit for gas. This limit is sufficient for EOAs and it helps protect the contract from re-entrancy attacks possibly coming from it, even though it is not recommended to rely solely on that. As a result, it used to be the preferable choice of transferring ETH.

However, if you want to allow other contracts to interact with your smart contract (e.g. considering account abstractions), this gas limit may not be sufficient.

Recommendation

Replace the usage of `transfer` with the `call.value` when transferring ETH to other accounts as the `call.value` does not have a limit on the used gas. Following checks-effects-interactions pattern is enough to protect from re-entrancy attacks.

Resolution

The issue was fixed according to our recommendation in the pull requests number 6. The fix contained a bug where the parentheses after the `call` were missing, thus causing Eth to not be transferred. The function was not invoked. We noticed this bug during the pull request review process and it was fixed in the pull request number 7.

FTEA-408 Code style issues

Category	Vulnerable commit	Severity	Status
Code Style	`26864b1ffb`	INFORMATIONAL	RESOLVED

Description

During our audit, we identified several code style issues that, while not directly impacting the security of the contract, could lead to confusion or potential issues in the future.

1. The variable `borrowerBondTokenId` does not semantically belong to the `LoanRequestData` struct anymore. The bond token is minted when a lender accepts the loan, so it has little meaning inside the loan's request data. We recommend moving this variable to the `Loan` struct instead. Additionally, the inclusion of the `borrowerBondTokenId` in the `LoanRequestCreated` event is unnecessary.
2. The visibility of the `loansCounter` variable is not explicitly set. It is a best practice to explicitly declare the visibility of state variables. Additionally, making it `public` helps tests know what id a new loan will receive.
3. The `claimCollateral` function does not strictly follow the checks-effects-interactions pattern. More precisely, the change of the `status` variable to `CLAIMED` should precede the potential burning of the bond token.

Recommendation

We recommend addressing these code style issues to improve the readability and maintainability of the contract.

Resolution

The issue was fixed according to our recommendation in the pull request number 8.

A Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the agreement between VacuumLabs Bohemia s.r.o. (Vacuumlabs) and FT Labs GmbH (Client) (the Agreement), or the scope of services, and terms and conditions provided to the Client in connection with the Agreement, and shall be used only subject to and to the extent permitted by such terms and conditions. This report may not be transmitted, disclosed, referred to, modified by, or relied upon by any person for any purposes without Vacuumlabs's prior written consent.

This report is not, nor should be considered, an endorsement, approval or disapproval of any particular project, team, code, technology, asset or anything else. This report is not, nor should be considered, an indication of the economics or value of any technology, product or asset created by any team or project that contracts Vacuumlabs to perform a smart contract assessment. This report does not provide any warranty or guarantee regarding the quality or nature of the technology analysed, nor does it provide any indication of the technology's proprietors, business, business model or legal compliance.

To the fullest extent permitted by law, Vacuumlabs disclaims all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. This report is provided on an as-is, where-is, and as-available basis. Vacuumlabs does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by Client or any third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services, assets and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Vacuumlabs will not be a party to or in any way be responsible for monitoring any transaction between you and client and/or any third-party providers of products or services.

This report should not be used in any way by anyone to make decisions around investment or involvement with any particular project, services or assets, especially not to make decisions to buy or sell any assets or products. This report provides general information and is not tailored to anyone's specific situation, its content, access, and/or usage thereof, including any associated services or materials, shall not be considered or relied upon as any form of financial, investment, tax, legal, regulatory, or other advice.

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Vacuumlabs prepared this report as an informational exercise documenting the due diligence involved in the course of development of the Client's smart contract only, and This report makes no claims or guarantees concerning the smart contract's operation on deployment or post-deployment. This report provides no opinion or guarantee on the security of the code, smart contracts, project, the related assets or anything else at the time of deployment or post deployment. Smart contracts can be invoked by anyone on the internet and as such carry substantial risk. Vacuumlabs has no duty to monitor client's operation of the project and update the report accordingly.

The information contained in this report may not be complete nor inclusive of all vulnerabilities. This report is not comprehensive in scope, it excludes a number of components critical to the correct operation of this system. You agree that your access to and/or use of, including but not limited to, any associated services, products, protocols, platforms, content, assets, and materials will be at your sole risk. On its own, it cannot be considered a sufficient assessment of the correctness of the code or any technology. This report represents an extensive assessing process intending to help Client increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology, however blockchain technology and cryptographic assets present a high level of ongoing risk, including but not limited to unknown risks and flaws.

While Vacuumlabs has conducted an analysis to the best of its ability, it is Vacuumlabs's recommendation to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring and/or other auditing and monitoring in line with the industry best practice. The possibility of human error in the manual review process is highly real, and Vacuumlabs recommends seeking multiple independent opinions on any claims which impact any functioning of the code, project, smart contracts, systems, technology or involvement of any funds or assets. Vacuumlabs's position is that each company and individual are responsible for their own due diligence and continuous security.

B Audited files

The files and their hashes reflect the final state at commit `95650b663baddbbf07d8fcf489b8a8e1473f7a32` after all the fixes have been implemented.

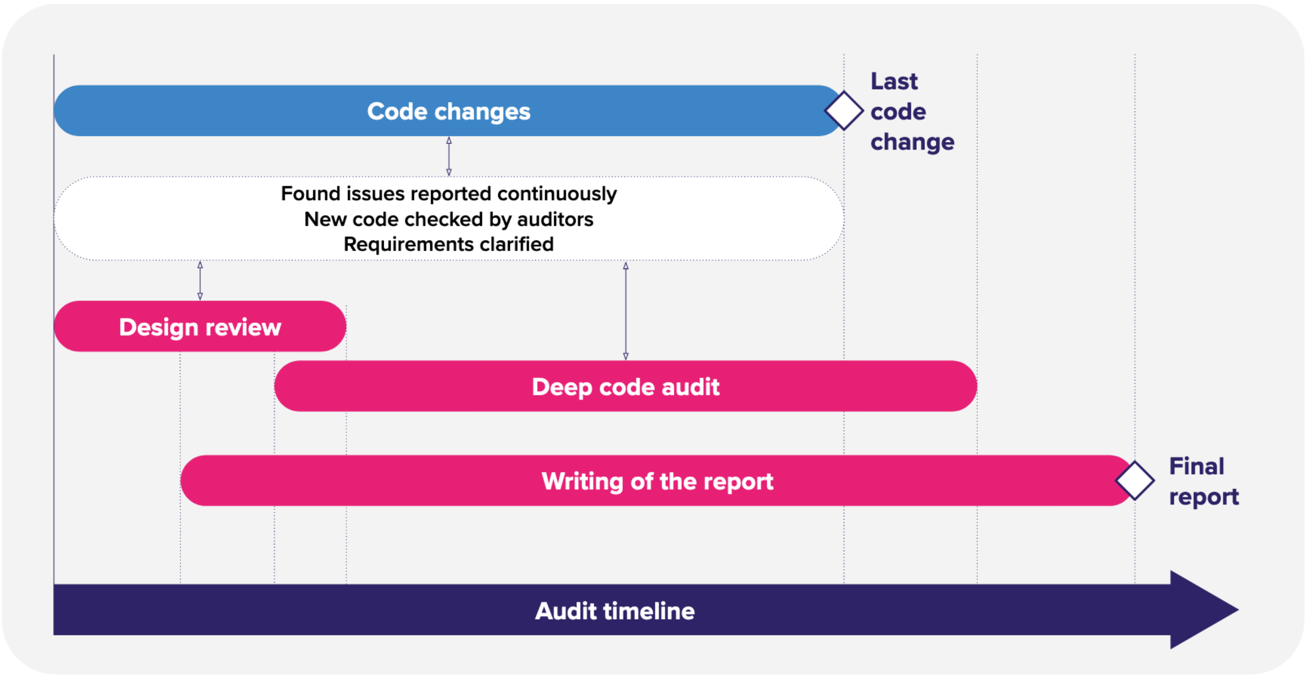
SHA256 hash	Filename
`0586b...55f92`	`ft-solidity-p2p-loans-EVM/bonds.sol`
`e38e5...d688f`	`ft-solidity-p2p-loans-EVM/ft_p2p.sol`

Please note that we did **not** audit other contracts located in other folders of the repository that are present at the final commit. This includes the OpenZeppelin libraries or the erc20.sol contract supposedly used only for testing purposes.

C Methodology

Vacuumlabs' agile methodology for performing security audits consists of several key phases:

1. Design reviews form the initial stage of our audits. The goal of the design review is to find larger issues which result in large changes to the code fast.
2. During the deep code audit, we verify the correctness of the given code and scrutinize it for potential vulnerabilities. We also verify the client's fixes for all discovered vulnerabilities. We provide our clients with status reports on a continuous basis providing them a clear up-to-date status of all the issues found so far.
3. We conclude the audit by handing over a final audit report which contains descriptions and resolutions for all the identified vulnerabilities.



Throughout our entire audit process, we report issues as soon as they are found and verified. We communicate with the client for the duration of the whole audit. During our audits, we check several key properties of the code:

- Vulnerabilities in the code
- Adherence of the code to the documented business logic
- Potential issues in the design that are not vulnerabilities
- Code quality

Ethereum audits

During Ethereum Virtual Machine (EVM) audits, we primarily use a manual approach to identify vulnerabilities and other issues. We are on a hunt for all Smart Contract Weaknesses[3], which include issues such as:

- Unencrypted private data on-chain
- Message calls with hardcoded gas amounts
- Unexpected Ether balances
- Requirement violations
- Missing protection against signature replay attacks
- Weak sources of randomness from chain attributes
- Signature malleability
- Transaction order dependence
- Delegatecall to untrusted callees
- Use of deprecated Solidity functions
- Reentrancy
- Floating pragma
- Outdated compiler versions
- Integer overflow and underflow

In addition to these well-known vulnerabilities, we also check the code for project-specific vulnerabilities, business logic flaws and game theoretic (lack of) incentives. We use several automated tools to expedite our audit; however, we do not solely rely on them and manually verify the results of those tools. The tools include but are not limited to:

- Slither
- Mythril
- Woke

We also provide the client with proof-of-concept tests that demonstrate the exploitability of the most critical issues or otherwise interesting or harder-to-verify issues.

[3]https://swcregistry.io/

D Issue classification

Severity levels

The following table explains the different severities.

Severity	Impact
CRITICAL	Theft of user funds, permanent freezing of funds, protocol insolvency, etc.
MAJOR	Theft of unclaimed yield, permanent freezing of unclaimed yield, temporary freezing of funds, etc.
MEDIUM	Smart contract unable to operate, partial theft of funds/yield, etc.
MINOR	Contract fails to deliver promised returns, but does not lose user funds.
INFORMATIONAL	Best practices, code style, readability, documentation, etc.

Resolution status

The following table explains the different resolution statuses.

Resolution status	Description
RESOLVED	Fix applied.
PARTIALLY RESOLVED	Fix applied partially.
ACKNOWLEDGED	Acknowledged by the project to be fixed later or out of scope.
PENDING	Still waiting for a fix or an official response.

Categories of issues

The following table explains the different categories of issues.

Category	Description
Design Issue	High-level issues in the design. Often large in scope, requiring changes to the design or massive code changes to fix.
Logical Issue	Medium-sized issues, often in between the design and the implementation. The changes required in the design should be small-scaled (e.g. clarifying details), but they can affect the code significantly.
Code Issue	Small in size, fixable solely through the implementation. This category covers all sorts of bugs, deviations from specification, etc.
Code Style	Parts of the code that work properly but are possible sources of later issues (e.g. inconsistent naming, dead code).
Documentation	Small issues that relate to any part of the documentation (design specification, code documentation, or other audited documents). This category does not cover faulty design.
Optimization	Ideas on how to increase performance or decrease costs.

E Report revisions

This appendix contains the changelog of this report. Please note that the versions of the reports used here do not correspond with the audited application versions.

v1.0: Main audit

Revision date:	2023-09-27
Final commit:	`95650b663baddbbf07d8fcf489b8a8e1473f7a32`

We conducted the audit of the main application. To see the files audited, see Audited files.

Full report for this revision can be found at url.

F About us

Vacuumlabs has been building crypto projects since the early days.

- We helped create WingRiders, currently the second largest decentralized exchange on Cardano (based on TVL).
- We are behind the popular AdaLite wallet. It was later improved into a multichain wallet NuFi.
- We built the Cardano applications for the hardware wallets Ledger and Trezor.
- We built the first version of the cutting-edge decentralized NFT marketplace Jam On Bread on Cardano with truly unique features and superior speed of the interface.

Our auditing team is chosen from the best.

- Talent from esteemed Cardano projects: WingRiders and NuFi
- Rich experience across Google, traditional finance, trading and ethical hacking
- Award-winning programmers from ACM ICPC, TopCoder and International Olympiad in Informatics
- Driven by passion for program correctness, security, game theory and the blockchain technology

We are a trusted Cardano ecosystem development partner

Contact us:

audit@vacuumlabs.com