FluidTokens Peer-to-Peer Loans v3

Audit Report v1

December 21, 2023

Contents

Revision table

Report version	Report name	Date	Report URL
1.0	Main audit	2023-12-21	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-based collateral*. The protocol can be initiated by either a person interested in borrowing (a borrower) or by a person interested in lending (a lender). 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 in this case.* 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.

Similar to a loan request, a person interested in lending and collecting interest on it (a lender) can also create a request. A collection offer request contains the resulting loan information, including an interest rate, accepted collateral options, a maximum size of a loan provided per a selected collateral option, a duration of the loan, and a number of installments in which the loan will be paid back. The whole amount to be lent is locked in the collection offer request.

Anyone can accept this collection offer request by taking a loan directly from the collection offer request provided they lock a sufficiently valuable collateral based on the parameters set out by the lender in the collection offer request. *The collateral in this case can contain multiple different NFTs from a single collection.* The borrower does not have to take the full loan amount. The remainder needs to be left untouched in the same collection offer request. Multiple loans can therefore be taken from the same collection offer. The lender can cancel the collection offer request before it is accepted.

If any single loan repayment is not paid on time, the lender can claim the collateral – the underlying NFTs. 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 a request (a loan request or a collection offer request) is accepted. 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 functionalities, 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 `f4dcdd8d9c813272d254dd4d1a3576faa326c0ae` and it lasted from 13 November 2023 to 21 December 2023. The timeframe is inclusive of periods in which we were awaiting the implementation of fixes by the client. We interacted mostly on Discord and gave feedback in GitHub pull requests. The team fixed all issues to our satisfaction, except for 2 minor and 1 informational findings that were acknowledged. They do not represent security threats to the system and can be mitigated by proper expectation management and communication.

The scope of the audit was limited to the smart contract files only. 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 some issues we found as part of this audit.

We performed a design review along with a deep manual audit of the code 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. See more about our methodology in Methodology.

The commit `d2157cc1e759259d25d41961a28cc042daabb2cd` 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 Aiken and the contracts are intended to run on Cardano. To avoid any doubt, we did not audit Aiken itself.

Summary of findings

During the audit, we found and reported: 3 critical, 2 major, 1 medium, 4 minor, and 6 informational findings. All findings were fully resolved except for these 3 that were acknowledged:

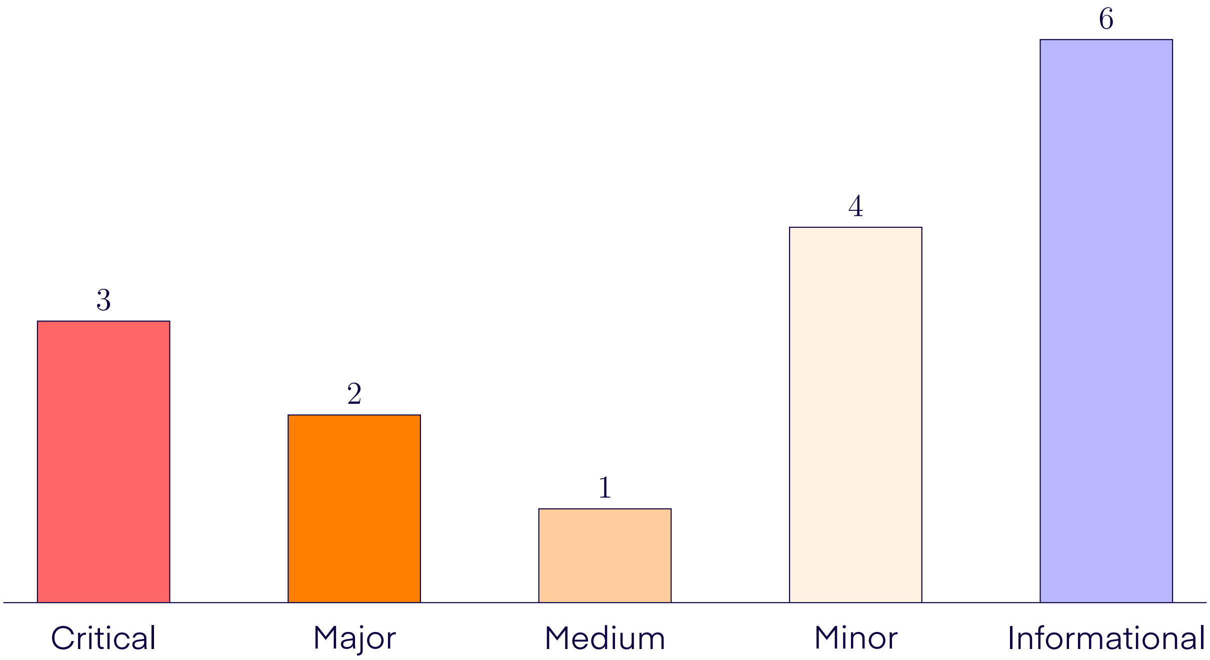
1. **FTA2-303**: Min Ada is not handled by the smart contract. As mentioned in the issue, the min Ada has no special handling in the smart contracts. The most notable downside is that loans with many installments require a min Ada downpayment in every repayment, possibly even on top of the loan amount. Clear communication is crucial here.
2. **FTA2-304**: Undefined repayments' staking credential. Since the bond ownership can change, it is not easy to make the repaid but unclaimed payments accrue Ada staking rewards on the bond holder's account. It is left unenforced in the smart contract. As a result, a repayer can possibly set it to any stake credential. Transactions coming from the official website will have the staking credentials set to the protocol's stake key.
3. **FTA2-405**: Undocumented assumptions and unchecked fields. This is an informational finding notifying the client not to rely on certain fields in the datums for the purposes of their web application as they can be set maliciously. More scrupulous documentation is encouraged.

The critical issues were of two kinds. The first finding arose from a code edge case – a possible overflow in the hash computation of the bond token name, allowing for multiple bonds with the same name. The other two findings were caused by an overlooked different parsing mechanism of certain inputs and outputs resulting in a lack of address validation.

The major finding FTA2-101 highlights the importance of resolving compiler warnings and maintaining a thorough test coverage. It involved two minting policies that were essentially identical, except for a field responsible for differentiating them that was not actively used in the code. The field was flagged in the Aiken warnings. It resulted in the policies being compiled into the same bytecode. Consequently, this affected certain essential functionalities in the rest of the code that assumed distinct policies.

The rest of the issues consisted of minor edge cases that the contracts did not handle and of code style suggestions to make the code more readable and less prone to errors.

2 Severity overview



Findings

ID	TITLE	SEVERITY	STATUS
FTA2-001	The same bond NFT can be minted multiple times	CRITICAL	RESOLVED
FTA2-002	Borrower can claim his collateral prematurely	CRITICAL	RESOLVED
FTA2-003	Borrower can steal the whole content of a collection offer pool	CRITICAL	RESOLVED
FTA2-101	Lender and borrower bonds use the same policy	MAJOR	RESOLVED
FTA2-102	Repayments are locked when active loan is claimed	MAJOR	RESOLVED

Continued on next page

ID	TITLE	SEVERITY	STATUS
FTA2-201	Double satisfaction in the loan amount payment	MEDIUM	RESOLVED
FTA2-301	Script hashes in the code are placeholders	MINOR	RESOLVED
FTA2-302	Duplications of type declarations	MINOR	RESOLVED
FTA2-303	Min Ada is not handled by the smart contract	MINOR	ACKNOWLEDGED
FTA2-304	Undefined repayments' staking credential	MINOR	ACKNOWLEDGED
FTA2-401	Aiken warnings	INFORMATIONAL	RESOLVED
FTA2-402	Helper functions are declared multiple times	INFORMATIONAL	RESOLVED
FTA2-403	Graveyard design improvement	INFORMATIONAL	RESOLVED
FTA2-404	Incorrect documentation of the loan request's redeemer	INFORMATIONAL	RESOLVED
FTA2-405	Undocumented assumptions and unchecked fields	INFORMATIONAL	ACKNOWLEDGED
FTA2-406	Naming and shadowing	INFORMATIONAL	RESOLVED

FTA2-001 The same bond NFT can be minted multiple times

Category	Vulnerable commit	Severity	Status
Code Issue	f4dcdd8d9c	CRITICAL	RESOLVED

Description

It is possible to mint multiples of the same borrower or lender bond NFTs split across different transactions. The reason is an overflow of the UTxO's index in the token name formula:

```
let tokenName = sha2_256(bytearray.push(utxo, index))
```

As the `bytearray.push`'s documentation mentions, when the given byte is greater than 255, it wraps around:

```
bytearray.push(#"0203", 1)
  == bytearray.push(#"0203", 257)
  == #"010203"
```

A sample attack can look like this:

1. A lender makes a huge transaction with multiple smaller outputs, say more than 257. The outputs now have the same transaction identifier and different indices.
2. The lender lends to a borrower and mints his lender NFT in the process. As a reference, he uses an UTxO he prepared – the one with the index equal to 1.
3. The lender sells his bond to somebody else for a decent price – after all, the buyer of the bond will be repaid the loan plus the interest.
4. The lender can now mint another lender NFT which is identical to the one he just sold. He can do so by referencing another UTxO he prepared – the one with the index equal to 257. Due to the overflow in the token name computation, the token name is not unique.
5. Owning the lender NFT, both the buyer of the bond and the attacker can now withdraw repayments. The attacker will likely watch it more closely and be the first one to withdraw.

Recommendation

We recommend checking that the index is between 0 and 255.

Resolution

The issue is resolved in the pull request number 1.

FTA2-002 Borrower can claim his collateral prematurely

Category	Vulnerable commit	Severity	Status
Logical Issue	`f4dcdd8d9c`	CRITICAL	RESOLVED

Description

There is no check for an ongoing active loan's address. As a result, a borrower can set the address of the ongoing active loan's UTxO when repaying the first installment to any dummy smart contract address controlled by him. Afterwards, he can freely claim his locked collateral. He is not obliged to repay the rest of the loan or the interest amount.

Recommendation

Make sure to add a check for the `activeLoanOutput`'s address to be equal to the `ownScriptHash`. You can add the check e.g. into the `validate_output_to_active_loan` function.

Resolution

The issue is resolved in the pull request number 1.

FTA2-003 Borrower can steal the whole content of a collection offer pool

Category	Vulnerable commit	Severity	Status
Logical Issue	`f4dcdd8d9c`	CRITICAL	RESOLVED

Description

Similar to the issue FTA2-002 with the active loan's address, there is no check for an ongoing collection offer pool's address. The collection offer pool verifies only the staking credential of its ongoing output, but not the payment credential.

An attacker can take an incomplete loan from the collection offer and then change the payment credential of the ongoing collection offer to his own script address. The result is a complete loss of the whole collection offer pool.

Recommendation

Add a check for the ongoing collection offer's payment credential into the collection offer validator.

Resolution

The issue was resolved in the pull request number 1.

FTA2-101 Lender and borrower bonds use the same policy

Category	Vulnerable commit	Severity	Status
Logical Issue	f4dcdd8d9c	MAJOR	RESOLVED

Description

The policy governing the lender and the borrower bonds is located in the `lender_bond.ak` and the `borrower_bond.ak`, respectively. The only difference between the files is the value of a variable called `bondType`. However, this variable is not used. That results in exactly the same policy hashes as can be verified in the `plutus.json` file that contains all the compiled hashes.

The other parts of the code assume that the policies are different, though. Since the policy does not allow for multiple asset names to be equal, looking at the loan request's `tokens_sent_to_lender_and_borrower` function, it becomes clear that it's impossible to lend to any loan request. It is impossible to mint two such tokens and it's required by the validator at the same time.

Recommendation

We recommend using the `bondType` variable as a parameter of the minting policies. Additionally, you could reuse the same code in the files and remove one file – since they are different only in the `bondType` variable.

Resolution

The policies are parametrized in the pull request number 1.

FTA2-102 Repayments are locked when active loan is claimed

Category	Vulnerable commit	Severity	Status
Design Issue	`f4dcdd8d9c`	MAJOR	RESOLVED

Description

When claiming an active loan, the validator checks that the lender's NFT is burned. Therefore, the NFT can not be used to withdraw any associated leftover repayment UTxOs anymore.

If the borrower repays some repayments, but then stops, the lender can liquidate the loan and claim his collateral. If, however, the lender first claims the collateral, he is not able to withdraw the already repaid repayments as he no longer owns the lender NFT.

Recommendation

We recommend not burning the lender's NFT.

Resolution

The issue was resolved in the pull request number 1.

FTA2-201 Double satisfaction in the loan amount payment

Category	Vulnerable commit	Severity	Status
Design Issue	`04ccbfee4c`	MEDIUM	RESOLVED

Description

In the current loan request scenario, the loan amount is paid directly to the borrower. There are safeguards against a double-satisfaction assuming multiples of the same scripts. However, it could potentially clash with another protocol that expects a certain payment to the same borrower. A malicious lender could exploit this by batching another protocol's operation (e.g. an NFT marketplace listing of the same borrower) with the loan request in the same transaction, leading to a double satisfaction issue – satisfying both the listing and the loan request.

Recommendation

One way to resolve this is to forbid those other scripts. If that is not the wanted course of action, we recommend not sending the loan amount directly to the borrower. Instead, consider sending it to a dedicated smart contract from where the borrower can claim it. This approach would be similar to a "claim" smart contract, akin to your existing "graveyard" smart contract. It would prevent potential cross-protocol double satisfaction issues on the loan amount. It is crucial that this dedicated smart contract is used exclusively for this protocol to avoid potential clashes. We even recommend adding a metadata parameter into the contract to avoid even random clashes – for example the name, the version and the purpose of the contract.

Resolution

The issue was resolved in the pull request number 2 by forbidding other script inputs in the relevant transactions. There could still be very rare situations in which double satisfaction is possible among this script and a minting policy or a reward script from another protocol. We explain such situations in our blogpost. However, it is okay to rely on that policy / reward script implementing double satisfaction prevention as well – checking that there is no script input. As a result, we consider this fix sufficient and this edge case improbable in practice.

FTA2-301 Script hashes in the code are placeholders

Category	Vulnerable commit	Severity	Status
Code Issue	f4dcdd8d9c	MINOR	RESOLVED

Description

The script hashes across the codebase such as the `lendersNftCs`, `borrowerNftCs`, `repaymentSCHash`, and many more are just placeholders with a constant value of `35b2...955e`. The value does not correspond to any value from the `plutus.json` file.

We assume that the values are just placeholders that are intended to be changed to the final hashes coming post-audit, hence the severity. However, to both keep track of this and since it is not deployable and testable in this state, we report it. The severity of the issue would be major if it was forgotten.

Recommendation

We recommend maintaining up-to-date hash references and testing all versions of the code. We also encourage an easier, more automatic way of updating them. You could do it in a build script, assuming you take the hashes as parameters.

Resolution

The issue was resolved in the pull request number 3 by taking the hashes as parameters of the validators.

FTA2-302 Duplications of type declarations

Category	Vulnerable commit	Severity	Status
Code Style	f4dcdd8d9c	MINOR	RESOLVED

Description

Many of the types declared in the smart contract files are declared multiple times. Examples of these are: `Asset`, `CollectionAmount`, as well as all the datums of various validators. The datums are the worst offenders here, as they are declared under different names – for example, the Datum from the `repayment.ak` file is equivalent to the `RepaymentDatum` from the `active_loan.ak`. In the `active_loan.ak`, the `Datum` is equivalent to the `ActiveDatum` in the other files.

The big issue with this is that any change in any datum has to be propagated into all the other places in the codebase, where this datum is used. The compiler will not catch a problem if there is one. As there are no tests right now, neither those will catch it, resulting in locked funds as the validator won't be able to properly parse its datum.

Recommendation

To increase the readability and the security of the code and to adhere to no code duplication best practices, we recommend separating all the type declarations that are reused into a new `types.ak` file which can be imported where needed.

Resolution

The issue was resolved in the pull request number 1.

FTA2-303 Min Ada is not handled by the smart contract

Category	Vulnerable commit	Severity	Status
Design Issue	04ccbfee4c	MINOR	ACKNOWLEDGED

Description

Every UTxO has to have a minimal amount of Ada (min Ada) inside it. For script UTxOs, this is usually around $1-2$ Ada. The lending smart contracts do not handle min Ada specifically. This has the biggest effect on repayments – in the worst case scenario, the borrower creates a repayment UTxO for each installment and has to therefore pay the lender an additional $totalInstallments \times minAda$ Ada. The more installments a loan has, the more the borrower has to pay to the lender.

There are also other less severe instances of this finding – for example, when borrowing from a collection offer, the borrower has to supply the min Ada for the active loan, but the last borrower does not have to do this as he consumes the collection offer output.

Recommendation

One of the cleaner solutions to cater the repayments could be to track the address of the repayer in the datum of a repayment and resend the min Ada back to him, ideally indirectly using a claim script to avoid the double satisfaction vulnerability. The other discrepancies are comparably smaller in impact. It is possible to explore those as well upon request.

Alternatively, you can clearly explain the users the side effects of loans with many installments and that it is expected that they will have to pay back that amount of Ada on top.

Resolution

The issue was acknowledged by the client.

FTA2-304 Undefined repayments' staking credential

Category	Vulnerable commit	Severity	Status
Design Issue	`04ccbfee4c`	MINOR	ACKNOWLEDGED

Description

The staking credential of Ada contained within the repayments is not currently checked in the smart contracts. This allows the borrower to set it as they wish. According to the client, it should be set to the lender's staking credential.

Recommendation

Given the transferability of the lender bond, it may not be easily possible to determine who currently holds the lender bond. We recommend setting the staking credential either to the original lender's staking credential, the protocol's credential or not to change the smart contract logic – leave it up to the borrower (or the front-end constructing the transaction). That would mean acknowledging that advanced borrowers could set it to their credentials, though.

Resolution

The issue was acknowledged by the client. They will prefill the protocol's staking credential to repayments from their off-chain.

FTA2-401 Aiken warnings

Category	Vulnerable commit	Severity	Status
Code Style	`f4dcdd8d9c`	INFORMATIONAL	RESOLVED

Description

Running `aiken check` outputs 122 warnings. They are all related to the code style – mostly listing unused imports, unused types, unused constructors and four instances of single `when` statements that can be rewritten in a nicer way.

Recommendation

We recommend fixing all of the warnings. Given the nature of the warnings, it will result in a removal of a lot of code. That improves the overall readability. You can either run `aiken check` directly and go case by case or you can use Aiken's VSCode extension to highlight those occurrences that need fixing directly in the code.

Resolution

The issue was resolved in the pull request number 3.

FTA2-402 Helper functions are declared multiple times

Category	Vulnerable commit	Severity	Status
Code Style	f4dcdd8d9c	INFORMATIONAL	RESOLVED

Description

Many helper functions are declared over and over across multiple files – for example functions such as: `is_nft_spent`, `validity_range_within_an_hour`, `get_outputs_to_sc`, `get_inputs_from_sc`, `must_be_signed_by` or `get_own_hash`. Sometimes, those functions are not even used – for example: `must_be_signed_by` or `validity_range_within_an_hour` in the `active_loan.ak` file.

This makes the code bloated and difficult to read. It also makes bugs very easy to introduce and hard to notice when changing the code.

Recommendation

We recommend creating a single file that contains all the helper functions and importing them into the validators.

Resolution

The issue was resolved in the pull request number 1.

FTA2-403 Graveyard design improvement

Category	Vulnerable commit	Severity	Status
Design Issue	f4dcdd8d9c	INFORMATIONAL	RESOLVED

Description

Currently, the smart contracts check that the lender bond tokens are moved to the graveyard after the last repayment is withdrawn. The graveyard is a fail-safe mechanism with two main functionalities:

1. It holds unusable bond tokens (cleanup).
2. The owner can withdraw the bond token from it (fail-safe mechanism).

The repayment smart contract requires the user to send the token to the graveyard when he withdraws the final repayment. This means that sometimes the user might have to send the token to the graveyard and then take it back – e.g. when he withdraws the final repayment before the other repayments – he may not be able to claim all of the repayments in a single transaction and this assumes withdrawing in an arbitrary order.

A simpler design could make sending the token to the graveyard optional and up to the front-end. The front-end can decide whether to send the tokens to the graveyard or back to the user depending on whether he claimed all of the repayments already or not. It does not need to be validated by the smart contract. The presence of the bond token needs to be validated, though.

Recommendation

You do not need to check whether the tokens are burned or sent to the graveyard. Instead, we recommend deciding this on the frontend. Either the user can decide to send the unused tokens to the graveyard or the frontend can automatically detect that the user has no more repayments to claim.

Resolution

The issue was partially resolved in the pull request number 1 and fully resolved in the pull request number 4.

FTA2-404 Incorrect documentation of the loan request's redeemer

Category	Vulnerable commit	Severity	Status
Documentation	`04ccbfee4c`	INFORMATIONAL	RESOLVED

Description

The documentation for the `Lend` redeemer in the loan request contains a typo. The field `lenderAddress` is described as the field where the **borrower** wants his bond NFT. This is incorrect as it is the **lender**'s address and it is the lender who wants his bond to be sent there.

Recommendation

We recommend correcting the documentation to accurately reflect that the `lenderAddress` field is the address where the lender wants his bond NFT to be sent.

Resolution

The issue was resolved in the pull request number 2.

FTA2-405 Undocumented assumptions and unchecked fields

Category	Vulnerable commit	Severity	Status
Documentation	`04ccbfee4c`	INFORMATIONAL	ACKNOWLEDGED

Description

There are several assumptions in the datums that are not documented. For instance, the repayment datum's `installmentsContained`, `installmentsPaidSoFar` and `isFinalRepayment` fields are not necessarily validated by the smart contract. They are validated for UTxOs created in the expected flow, by interacting with the active loan. However, an attacker could create a repayment UTxO directly and put any values into the datums. Same applies to the `containedAmount` field in the collection offer's datum.

Additionally, the graveyard's `owner` needs to be a public key hash owner, not a smart contract. The same applies to the generic lender/borrower functionality, where the user must be able to sign transactions to claim repayments or cancel loan requests. If a smart contract address is used instead, it could result in locked funds.

Finally, if the collateral contains Ada, the Ada is unchecked by the smart contract and can be hence stolen from the loan request.

Recommendation

We recommend documenting these assumptions and ensuring that the front-end is robust against these issues and validates that the fields correspond with reality. Also, front-end checks should be implemented to prevent smart contract addresses from being used in these fields as well as not allowing the placement of Ada collateral.

Resolution

The issue was acknowledged by the client. They will address it off-chain.

FTA2-406 Naming and shadowing

Category	Vulnerable commit	Severity	Status
Code Style	`04ccbfee4c`	INFORMATIONAL	RESOLVED

Description

Naming improvement suggestions include:

1. Collection offer datum's `maxLoanAmnt` and `wantedCollections` do not explain what they represent. The `maxLoanAmnt` is not the maximum loan amount but rather the maximum at which a single collection multiple is enough as a collateral. As for the `wantedCollections`, a more suitable name could be `collateralOptions`. The `maxLoanAmnt` could then be renamed to something like `maxLoanPerSingleCollateralOption`.
2. There are a few places where variables are simply named `a`. This is acceptable as long as it is contained within a very simple function with a very simple logic and ideally a single usage of the variable. However, as we are talking about a smart contract logic which is critical, it would help to have more descriptive names. In the `input_is_included` function, the variable `a` is also assigned value twice and so it shadows the function-level variable `a`. We recommend renaming all variables named `a`.
3. A statement like `Some(someInput)` extracts the value of the option variable into the `someInput` variable. However, it asserts that the `Option` is of `Some` type and that it has a value that is then assigned to the `someInput` variable. It is therefore not a `someInput` anymore, but rather an `input`.
4. In the `input_is_included` function, the variable named `waited` could be renamed to e.g. `wanted` to better reflect its purpose.

Recommendation

The recommendation is part of the bullet points.

Resolution

The bullet points were mostly addressed in the pull request number 2 with the final fix in the pull request number 4.

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

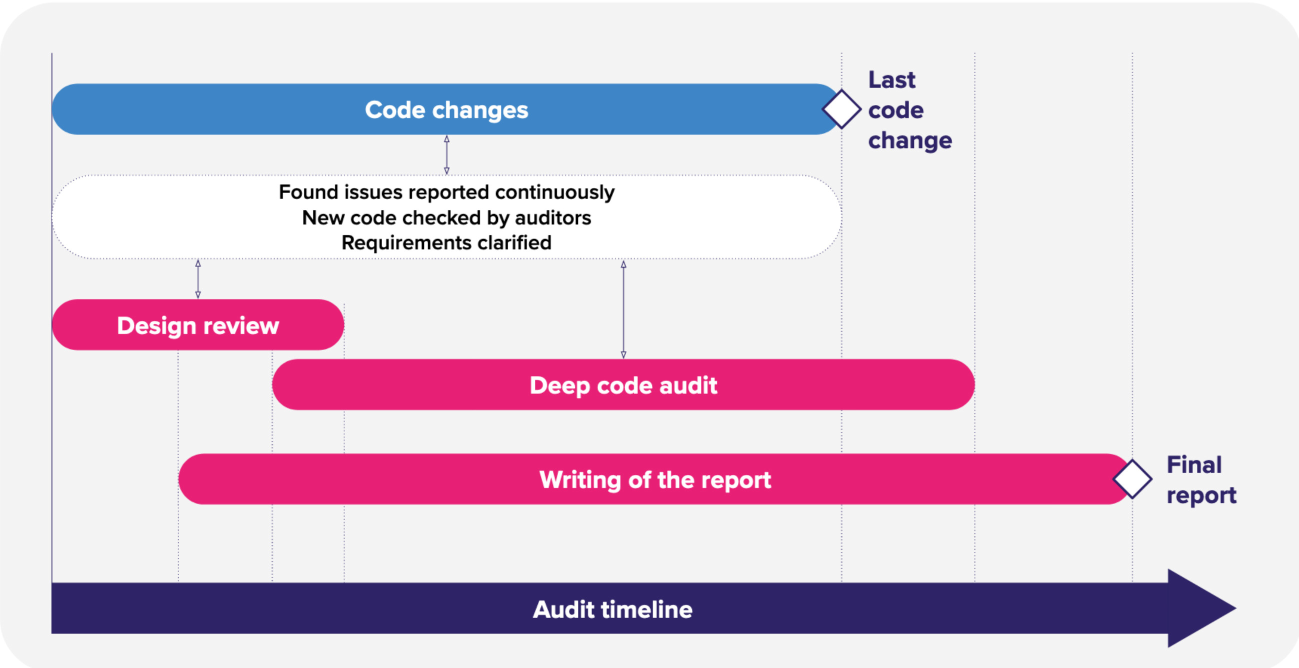
The files and their hashes reflect the final state at commit `d2157cc1e759259d25d41961a28cc042daabb2cd` after all the fixes have been implemented.

SHA256 hash	Filename
9679e...dcf22	lib/types.ak
8d811...a141c	lib/utils.ak
0170a...eb534	validators/active_loan.ak
9b589...bb8e9	validators/bond.ak
fb76e...714f0	validators/bonds_graveyard.ak
38e78...7a0b9	validators/collection_offer_pool.ak
9c53f...227a0	validators/loan_request.ak
9022b...b0c44	validators/repayment.ak

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:

1. Vulnerabilities in the code
2. Adherence of the code to the documented business logic
3. Potential issues in the design that are not vulnerabilities
4. Code quality

During our manual audits, we focus on several types of attacks, including but not limited to:

1. Double satisfaction
2. Theft of funds
3. Violation of business requirements
4. Token uniqueness attacks
5. Faking timestamps
6. Locking funds indefinitely
7. Denial of service
8. Unauthorized minting
9. Loss of staking rewards

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: 2023-12-21
Final commit: `d2157cc1e759259d25d41961a28cc042daabb2cd`

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

Full report for this revision can be found at url.

F About us

Vacuumlabs has been building crypto projects since the early days.

1. We helped create WingRiders, currently the second largest decentralized exchange on Cardano (based on TVL).
2. We are behind the popular AdaLite wallet. It was later improved into a multichain wallet NuFi.
3. We built the Cardano applications for the hardware wallets Ledger and Trezor.
4. 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 both the interface and transactions.

Our auditing team is chosen from the best.

1. Talent from esteemed Cardano projects: WingRiders and NuFi
2. Rich experience across Google, traditional finance, trading and ethical hacking
3. Award-winning programmers from ACM ICPC, TopCoder and International Olympiad in Informatics
4. 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