TWEAG
by Modus Create

Technical Review of Cerra P2P Lending

High Assurance Software Group
February 13, 2024

Contents

Chapter 1

Executive Summary and Scope

This report is presented without warranty or guaranty of any type. This report lists the most salient concerns that have so far become apparent to Tweag after a partial inspection of the engineering work. Corrections, such as the cancellation of incorrectly reported issues, may arise. Therefore Tweag advises against making any business decision or other decision based on this report.

Tweag does not recommend for or against the use of any work or supplier referenced in this report. This report focuses on the technical implementation provided by the project's contractors and subcontractors, based on their information, and is not meant to assess the concept, mathematical validity, or business validity of Cerra's product. This report does not assess the implementation regarding financial viability nor suitability for any purpose.

Scope and Methodology

Tweag looks exclusively at the on-chain validation code provided by Cerra. This excludes all the frontend files and any problems contained therein. Tweag manually inspected the code contained in the respective files and attempted to locate potential problems in one of these categories:

a) Unclear or wrong specifications that might allow for fringe behavior.

b) Implementation that does not abide by its specification.

c) Vulnerabilities an attacker could exploit if the code were deployed as-is, including:

- race conditions or denial-of-service attacks blocking other users from using the contract,
- incorrect dust collection and arithmetic calculations (including due to overflow or underflow),
- incorrect minting, burning, locking, and allocation of tokens,
- authorization issues,

d) General code quality comments and minor issues that are not exploitable.

Where applicable, Tweag will provide a recommendation for addressing the relevant issue.

Chapter 2

Audit

2.1 Methodology

Tweag analyzed the validator scripts comprising Cerra P2P Lending, contained in the repository[1] as of commit `4860578`[2]. The names of the files considered in this audit and their `sha256sum` are listed in Table 2.1.

`sha256sum`	File Name
`cccbd55...0ac00c9`	`Cerra/Lending/Contract/Lending/OnChain.hs`
`391b256...2074aed`	`Cerra/Lending/Contract/Lending/Types.hs`
`37bad56...2678891`	`Cerra/Lending/NFT/Borrower/OnChain.hs`
`adb1fa1...b4fdae4`	`Cerra/Lending/NFT/Borrower/Types.hs`
`aeb21df...90037dd`	`Cerra/Lending/NFT/Factory/OnChain.hs`
`1b13993...4e4b6c3`	`Cerra/Lending/NFT/Factory/Types.hs`
`6c79533...a64a1b4`	`Cerra/Lending/NFT/Lender/OnChain.hs`
`1b3d7d7...b16263b`	`Cerra/Lending/NFT/Lender/Types.hs`
`1c7e94a...c83cc37`	`Cerra/Lending/Oracle/Cerra/OnChain.hs`
`1eff80b...993b577`	`Cerra/Lending/Oracle/Cerra/Types.hs`
`6fd5b81...bd42b49`	`Cerra/Lending/Oracle/Orcfax/OnChain.hs`
`5abfa19...bd86f51`	`Cerra/Lending/Oracle/Orcfax/Types.hs`
`4fe5451...712b7c1`	`Cerra/Lending/Utils/Debug.hs`
`92b575a...fc90a01`	`Cerra/Lending/Utils/OnChainUtils.hs`
`fa16331...137aab6`	`Cerra/Lending/Utils/Settings.hs`
`ed2ddf6...9146bf3`	`Cerra/Lending/Utils/Utils.hs`

TABLE 2.1: *On-chain code source files and their `sha256sum` that were analyzed as part of the review*

Our analysis is based on the documentation provided by Cerra, and on weekly meetings with Cerra. The relevant documentation files and their contents will be referred to as *the specification* of the protocol.

In order to experiment with Cerra P2P Lending and carry out test cases using our auditing library (cooked-validators), we had the script validator and minting policy go through our own Plutus compilation workflow rather than the default provided chain using plutonomy. These alterations are limited and do not affect a single line of the validators logic (in `... -> Bool` functions), data types, and helpers. They make it possible to interface cooked-validators with Cerra P2P Lending without any significant loss of time.

[1] `https://github.com/cerraio/lending-plutus`
[2] Full commit hash: `48605787f534d5acb336d85dfb4115323d90ddb9`

2.2 Findings

Table 2.2 lists our concerns with the current Cerra P2P Lending implementation based on our partial exploration during a limited period of time. Throughout the rest of this section, we will detail each of our findings.

Severity	Section	Summary
Critical	2.2.1.1	All assets can be stolen from lending UTxOs.
High	2.2.1.2	Loans can be liquidated before their deadline.
High	2.2.1.3	Lender can increase the amount of due interests.
Medium	2.2.2.1	The amount of Ada in the script is too strict.
Medium	2.2.2.2	Redeemers should reflect each distinct action.
Medium	2.2.2.4	Most of the contract logics should appear in the lending script.
Medium	2.2.2.5	Loans using Ada as asset are treated in a convoluted way.
Medium	2.2.2.6	Inputs other than the script ones should be allowed to have datums.
Medium	2.2.4.1	Validation guards are not centralized.
Medium	2.2.4.2	Bang patterns and laziness are use in an erratic manner.
Low	2.2.2.3	Borrower NFT could be referenced when retrieving loan assets.
Low	2.2.3.1	The factory NFT still requires offchain verifications.
Low	2.2.3.2	Oracle reference outputs are unspecified.
Low	2.2.4.3	Function names are sometimes misleading.
Low	2.2.4.4	The overall code quality and use of Haskell can be improved.
Lowest	2.2.3.3	Time units are underspecified and heterogeneous in the lending datum.

TABLE 2.2: *Table of findings*

2.2.1 Vulnerabilities

2.2.1.1 All assets can be stolen from lending UTxOs.

Severity: Critical

The lending script is vulnerable to datum hijacking. When a lending script UTxO is redeemed with "accept" and a new continuing UTxO with updated datum and/or value is supposed to be paid at the same script address, it is possible to actually trick the validators by paying to another address (script or pubkey) as long as we provide the datum that should have gone to the lending script. This means anyone can redirect the assets (Ada, loan assets, collateral, and tokens) to any address.

Carrying out the illegal action In an "accept" transaction that is supposed to create an updated UTxO for a lending position (such as "accept"), modify the destination address of where the lending datum and assets are paid to any desired adversary address, e.g. the pubkey address of a thief.

Cause of the issue The lending script validator with the "accept" redeemer only checks conditions about the tokens. The actual logic of the "accept" action is implemented in the lender minting policy, which is not aware of the address of the lending script.

This minting policy performs two major checks related to the expected continuing output for the lending script:

- It searches for the script output using `getContractOutput` (defined in `Cerra.Lending.Utils.Utils`) which goes through all the outputs and filters those with a datum according to `mustFindScriptDatum` (defined in `Cerra.Lending.Utils.OnChainUtils`). However, this helper function only guarantees that a UTxO carries a datum but any script or pubkey UTxO can hold a datum of type **`LendingDatum`**.
- It checks that the new datum has the right value after accepting a request, but any adversary pubkey output can include a datum with the right values.
- It checks that the total value of the assets in the obtained script output matches the expectations after accepting a request.

However it does not:

- Check the address of the output UTxOs to ensure assets are sent to the legitimate lending script.

Remediation We suggest to check that destination address of the assets indeed belongs to the lending script in the validator of the script itself. In module `Cerra.Lending.Utils.Utils`, function `ownContractOutput` does this. Additionally, to simplify the overall design and optimize validation costs, a common design pattern consists in requiring the continuing output to be the first output of the transaction. This avoids the need for a search through all output UTxOs.

Example traces Traces emphasizing this behaviour are prefixed with `datumHijackingAttack`. They showcase redirection of outputs in both accept cases (borrower and lender) to both possible targets (another script, or a private key).

2.2.1.2 ■ Loans can be liquidated before their deadline.

Severity: High

Lenders can liquidate any of their loan before the deadline has elapsed. Therefore, they can steal the collateral assets of borrowers who accepted these loans even when:

- they still have enough time to repay the withdrawn assets,
- the value ratio between collaterals and lent assets is still below the threshold for premature liquidation.

Carrying out the illegal action To liquidate before the deadline, the lender only has to provide an upper bound for the transaction's validity range that is set after the deadline. The lower bound is not checked and can be arbitrarily low. Hence, the validity range may include any time period between the current slot and the deadline slot. The rest of the transaction is similar to a legitimate liquidation after deadline using redeemer 4 for the lending script, and burning the tokens.

Cause of the issue Liquidation implies burning the lender token. The cause of the issue is in the minting policy for lender tokens, in `validateBurn` in module `Cerra.Lending.NFT.Lender.OnChain`. The condition that checks if the deadline has elapsed is `if loanEndsAt < nowTime` where `loanEndsAt` is computed correctly but `nowTime` is defined as the upper bound of the validity range (`getUpperBound (txInfoValidRange info)`) instead of the lower bound.

Remediation For liquidation, we suggest that the upper bound of the validity range be compared against the deadline instead. This way, the whole validity range would be guaranteed to be after the deadline. In addition, as a general recommendation, we think it is important to always remember that the time context of a transaction is a validity range from one slot to another and reflect this explicitly in design and implementation of validators. In particular, there is no concept of real time "now" in a transaction context. In our experience, although introducing real instant time (such as the `nowTime` variable) simplifies expressing the logic of the validators, it is actually often prone to errors down the line.

Example traces Trace `liquidateBeforeDeadline` showcases this behaviour.

2.2.1.3 Lender can increase the amount of due interests.

Severity: High

When a lender accepts a borrow offer placed in the lending script, they can set an arbitrarily early starting date in the datum of the loan. The borrower has already placed their collaterals in the script and will need to pay interests for much longer than they should if they want to retrieve them. For instance, if the loan was accepted at $t = 100$ by then lender while in fact placing $t = 0$ in the datum, the borrower will have to pay for 100 more time units of interests.

Carrying out the illegal action To carry out the illegal action, the lender has to accept a borrow offer and place any date prior to the current time both in the datum and as a lower bound of the validity range of the transaction to enjoy higher interests when repaid.

Cause of the issue In the minting policy for the lender token, when minting the token in the accept case, a check is made that the loan start time in the datum is in fact the lower bound of the validity range of the transaction. While this is correct in the case of a borrower accepting a loan offer, this is not correct the other way around. More precisely, we can read:

```
getLowerBound (txInfoValidRange info) == fromJustCustom loanStartTimeOut
```

Remediation Instead of ensuring that the loan start date corresponds to the lower bound of the transaction, the check should be made on the upper bound, thus leading to the following code:

```
getUpperBound (txInfoValidRange info) == fromJustCustom loanStartTimeOut
```

Example traces This behaviour is showcased in trace `startsAt0Attack`.

2.2.2 Design Flaws

2.2.2.1 The amount of Ada in the script is too strict.

Severity: Medium

Scripts in the product require a strict amount of Ada in the script outputs (either 2 or 4 Ada when the lending asset is not Ada itself). It is unclear why the requirement is there in the first place. In addition, since Plutus version 2, the minimal amount of Ada in an output is no longer two, but directly depends on the size a the UTxO. That is, there could be cases where the limitation of exactly 2 Ada prevents users from using the product, if the required amount of Ada is greater than 2. The way this minimal amount of Ada is computed is detailed here: `https://github.com/IntersectMBO/cardano-ledger/blob/master/doc/explanations/min-utxo-alonzo.rst`.

Remediation We suggest to remove the monitoring of the exact amount of ada present in the script. If somehow these checks are used to infer in which case we stand, we suggest to apply remediation from section 2.2.2.2 to avoid relying on such contextual information to make decisions.

2.2.2.2 Redeemers should reflect each distinct action.

Severity: Medium

Redeemers are specifying for what reason the user is either consuming a script UTxO or minting/burning a certain token. Each redeemer should be associated to a specific action, and thus lead to performing a certain set of checks associated with this behaviour. In Cerra P2P Lending codebase, redeemers encompass several actions at once, and the context is used to discriminate between such possible actions. This is an anti-pattern. Redeemers should be used to know which checks to be performed, and not the other way around. This is the case for instance in the lender and borrower minting policy, where a `TxOutRef` is passed as a redeemer in all cases, regardless of why we mint or burn such token. This is also for instance the case when liquidating a loan, which can happen for two distinct reasons that should have their distinct redeemers.

Remediation One redeemer should be created for each dedicated actions. More precisely, each subfunction in every script should be associated with a specific redeemer. This way, the context would not be used to decide which branch to go to, and instead, the context would be checked for validity regarding the current redeemer. In addition, redeemers should be defined as data types instead of, for instance, using plain integers. This would make for a more human readable codebase and would yield less area for errors, while not relying on partial information to make decisions.

2.2.2.3 Borrower NFT could be referenced when retrieving loan assets.

Severity: Low

When a borrower retrieves their assets after a lender has accepted their borrower offer, there is no minting nor burning involved. Instead, the borrower just shows their NFT in an input and then gets it back as an output. This needlessly enlarges the size of this transaction.

Remediation We suggest to use reference inputs in this specific transaction for the borrower to simply show their NFT instead of having it consumed given back to them.

2.2.2.4 Most of the contract logics should appear in the lending script.

Severity: Medium

Cerra P2P Lending is composed of 3 minting policies and 1 script, deployed as follows:

a) The lender and borrower minting policies are deployed from static data.

b) The lending script is deployed from the lender and borrower minting policies (thus knowing their associated currency symbols).

c) The factory minting policy is deployed from all of the above, thus being aware of both currency symbol and the lending script address.

Reading this list from bottom to top gives us indication as to which validation element possesses the most knowledge, and thus should yield the most checks. In practice though, most of the verification is done in the lender and borrower minting policy, which have the less knowledge about the other components of the contract. This is, for instance, what most likely caused 2.2.1.1.

Remediation The factory minting policy has the wider knowledge, but is only called at the very beginning of the life cycle of the product and at the very end so it is not a very reliable verification entity (see 2.2.3.1). The lending script however is aware of both the lender and the borrower minting policy and is called at most steps of the life cycle of a loan, thus making it the perfect candidate to perform most of the verification. We thus suggest to move as many verification functions as possible to the lending script validator. If the validator becomes too big when compiled, we suggest to attach it to a certain UTxO as a reference script and have the offchain code pass the UTxO as a reference input.

2.2.2.5 Loans using Ada as asset are treated in a convoluted way.

Severity: Medium

When using Cerra P2P Lending, any assets can be lent, including Ada. Although it seems that lending Ada is just a special case of an otherwise wider logics, it turns out that it yields quite a different validation process. Indeed, there are several places in the code where the validation involves counting the number of different assets present in a certain value. In those case, this number varies depending if Ada are exchanged or not, because there will always be Ada in a UTxO, but there will not always be an additional asset. This also changes the checks about the actual number of Ada in the UTxO (which we discourage, see 2.2.2.1). In practice, this difference is handled in an obscure manner within some helper functions that arbitrarily change a result based on the nature of the exchanged asset. Here is an example:

```
lovelaceAmount :: AssetClass -> Value -> Integer
lovelaceAmount asset val = case asset == adaCoin of
  True -> 2_000_000
  False -> getLovelace (fromValue val)
```

This is a convoluted and error-prone way of handling this difference in behaviour.

Remediation We strongly discourage to push into the leaves of the code (typically helper functions) this kind of high level case distinction. Instead, we suggest to have, in each validation function, a set of guards common to each kind of asset, and then a branching on the nature of then asset leading to a different set of relevant guards for Ada or non-Ada assets.

2.2.2.6 Inputs other than the script ones should be allowed to have datums.

Severity: Medium

The logic of many validation functions requires to find in the input of the transaction a UTxO with the script datum. To retrieve this input, the code is as follows:

```
contractInput :: TxInInfo
!contractInput = case [i | i <- txInputs, scriptDatumExists (txInInfoResolved i)] of
  [i] -> i
  _ -> debugError "E15" True
```

This code snippet goes through all the inputs of the transaction, and selects the only one of them that has a datum. Not only this does not ensure that the input is at the right address and has the right kind of datum (see 2.2.1.1) but it also prevents any other input in the transaction from having some kind of a datum. This is problematic because it is often encouraged that scripts which pay to a private key do it with a datum (often containing a certain unique field) to avoid double satisfaction vulnerabilities. In the current state of the product, a user who possesses assets in a UTxO with a datum would be unable to use them with the product, unless they make a preliminary transaction to spend this UTxO and pay the value to a datumless UTxO at their own address first.

Remediation This remediation element should be handled as part of a greater change to solve 2.2.1.1 and 2.2.2.4. However, if it had to be solved purely in the helper function displayed earlier, we suggest to look for a datum of a specific kind instead of any.

2.2.3 Unclear Specification

2.2.3.1 The factory NFT still requires offchain verifications.

Severity: Low

In Cerra P2P Lending, lending script UTxOs carry a factory token minted at loan or borrow request creation, and eventually burnt at the end of the request lifetime (cancellation, collection, liquidation).

The lending validator ensures this token is preserved in each transaction until burning. As for the minting policy of factory tokens, it ensures they are placed in a lending script UTxO.

There are common guarantees and features one usually expects from the widespread family of "NTF in script UTxO" design patterns. However, the specification of Cerra P2P Lending does not make it clear which are applicable here and which are not. In particular here, onchain verifications are not self sufficient.

Recap of the design pattern In contracts such as Cerra P2P Lending, script UTxOs are spent and produced again in the same transaction with slight alterations accounting for the desired update following an action. Script validators ensure that each link of the chain of actions is legit. However, they can only restrict how UTxOs are spent but it is always possible to pay any value and any datum to a script. This means the initial datum and values can't be checked that way. To ensure the initial datum and values are correct, and in turn the rest of the chain of actions, scripts usually require to carry a validity NFT. The minting policy of the validity NFT usually checks that the minted token goes to a script UTxO with the right initial datum and value. The script validator then ensures the NFT is forwarded at each step and eventually burnt, hence solving the initialization problem. Additionally, the validity NFT minting policy sometimes requires to pay an extra fee to the product company pubkey (what is done here in the lender or borrower token minting policy instead).

Dependency cycle challenge The script validator must know about the NFT's currency symbol (hash of its minting policy) and the NFT's minting policy must know about the script address (hash of the script validator). While there is not direct way to address this dependency cycle, there compromises and workarounds.

Cerra's design choice In Cerra P2P Lending, the lending script validator doesn't know about the Cerra's factory NFT minting policy, hence breaking the dependency cycle. In order to identify the factory token in a given UTxO, the lending script processes by elimination (ruling out Ada, loan assets, collateral assets, and lender/borrower tokens).

Consequences If a loan or borrow request is initialized with a token that does not have the factory currency symbol, the product including script and minting policies will work as usual: accepting, withdrawing, repaying, and liquidating will be allowed by the validators. As a result, the factory token is no guarantee that **all** UTxOs sitting at the lending script address are trustworthy and safe to redeem. The NFT should be seen instead as a label asserting whether a loan/borrow request has been initialized following Cerra's expectations. This means trust is moved out of the decentralized Cardano blockchain and put in offchain tooling which needs to filter out bad loan/borrow requests.

Remediation We suggest to make these design choice implications crystal clear in the specification of the product. Besides, offchain tooling should be audited and secured accordingly.

Example traces Trace `wholeLifetimeWithDummyToken` showcases the use of the product with a different Factory token.

2.2.3.2 ■ 0racle reference outputs are unspecified.

Severity: Low

When liquidating a loan because of ratio threshold, it is expected to pass the 2 oracle UTxOs as reference inputs. In the code, we can see that they are expected to lie at the first 2 positions of the reference inputs list, but this is not specified. In particular, if the script is passed as a reference script, this could alter the list and thus make the transaction not validate.

Remediation We suggest to document this restriction, or to look for the two UTxOs in the list without constraining them to be the first two elements.

2.2.3.3 ■ Time units are underspecified and heterogeneous in the lending datum.

Severity: Lowest

In the datum of the lending script, we can find the following field related to time:

- `scLoanStartTime`, expressed in POSIXTime (or ms)
- `scLoanLength`, expressed in s
- `scInterestPerSecond`, expressed in $10^{-12}\ s^{-1}$

This is heterogeneous and can be confusing.

Remediation We suggest to either stick to the same unit, use well-known units such as epochs, or document the relation between them and the formula to compute interests properly.

2.2.4 Code Quality

2.2.4.1 ■ Validation guards are not centralized.

Severity: Medium

Typically, validators will follow the usual pattern where the overall validation functions are broken down into sub-functions based on the nature of the redeemer or some other relevant set of checks. The sub-functions are usually composed of two distinct parts: a series of assignments, based on the datum and the context of the transaction, and then a set of checks made over those assigned variables.

Having two distinct parts is extremely useful because it makes for a centralized set of verifications to be performed over the current input or the current minted sub-value. It makes it easy to assess what are the exact requirements for the transaction to be validated, and what are the values on which these requirements operate.

While the Cerra P2P Lending codebase follows this pattern in appearance, it is broken by the fact that most of the assignments are themselves the results of calls to partial functions (usually in the utility module) that will fail depending on a set of criteria. In practice, this means that the actual set of requirements is spread out over several places in the codebase. The programmer might thus lose visibility on the overall consistency of the validator, thus increasing the likelihood of mistakes.

Remediation We suggest to regroup the various validation criteria at the end of each validation subfunction, and to reframe all utility functions to be total.

For example, the following code snippet extracted from the utility module:

```
getContractInput :: TxInfo -> TxOut
getContractInput info = txInInfoResolved contractInput
  where
    !txInputs = txInfoInputs info
    contractInput :: TxInInfo
    !contractInput = case [i | i <- txInputs, scriptDatumExists (txInInfoResolved i)] of
      [i] -> i
      _ -> debugError "E15" True
```

... would become:

```
getContractInputs :: TxInfo -> [TxOut]
getContractInputs info = map txInInfoResolved contractInputs
  where
    !txInputs = txInfoInputs info
    !contractInputs = [i | i <- txInputs, scriptDatumExists (txInInfoResolved i)]
```

... and they would be used in the validation script as follows:

```
let inputs = getContractInputs txInfo in traceIfFalse "E15" (length inputs == 1) && P (head inputs)
```

... where `P` is a certain predicate over the input.

2.2.4.2 Bang patterns and laziness are use in an erratic manner.

Severity: Medium

Haskell is a lazy language, where values assigned to variables will only be evaluated when the variable is actually used. To circumvent this original behaviour and enforce evaluation of variables at declaration, Cerra P2P Lending codebase is scattered with variable assignments using the bang patterns language extension. However, this is often unclear why this pattern is used in some cases, and why it is not in others. In addition, laziness is sometimes used to actually postpone evaluation based on some guards later on in the validation function, thus making it extremely unclear what checks are performed in each cases, as these assignment can result in errors.

Remediation This issue is related to 2.2.4.1 in the sense that it hides to the reviewer and the developer the overall picture on which guards correspond to which checks. We advise to centralize all checks in guards and not rely on laziness to distinct several cases. For example, if a variable `a` is only used in one case, it should be defined within the scope of this case, and not before. We also advise to document the use of bang patterns or remove them altogether. This will make for clearer conditions and guards.

2.2.4.3 ■ Function names are sometimes misleading.

Severity: Low

The codebase contains occurrences of functions that are named in a misleading fashion. Although, this is a minor concern, having functions named in such a way that it creates wrong assumptions in the programmer's head is dangerous. An example is the following function (which relates to 2.2.2.5):

```
assetLength :: AssetClass -> Integer -> Integer
assetLength asset count = case asset == adaCoin of
  True -> count + 1
  False -> count
```

Although this function is named `assetLength`, it does not return the number of assets when the asset is Ada. This is misleading. Another example is the `mustFindScriptDatum` function, which definitely not guarantee to find something belonging to the script.

Remediation The names should be changed to properly match the actual semantics of the function instead of what the programmer *expects* the function to accomplish in the context of their view on the problem at hand.

2.2.4.4 ■ The overall code quality and use of Haskell can be improved.

Severity: Low

There are multiple instances in the codebase where good Haskell and coding practices overall could be used to improve the quality of the product and decrease the likelihood of bug, the size of the code, and the readability of the product. This concern lists some of those instances.

- The pattern `if a then True else False` is used. This in an anti-pattern, use a instead.
- The pattern `fromJust (Just value)` is used. Use `value` instead.
- List comprehension is overused and hurts readability. Use `filter` instead.
- Use pattern matching whenever possible. For instance:

  `OutputDatum dat -> PlutusTx.unsafeFromBuiltinData (getDatum dat)`

  can be replaced by

  `OutputDatum (Datum dat) -> PlutusTx.unsafeFromBuiltinData dat`.
- Helper functions can often be replaced by pattern matching, such as `getCS` and the like.

Combining the above suggestions, here is an example of code improvement. Consider the function:

```
isNFTBurned :: [(CurrencySymbol, TokenName, Integer)] -> CurrencySymbol -> Bool
isNFTBurned flattenVal symbol = case [ c | c <- flattenVal, symbol == getCS c
                                                              && getAmount c == -1
                                     ] of
  [_] -> True
  _ -> False
```

It could be, for instance, reframed as:

```
isNFTBurned :: [(CurrencySymbol, TokenName, Integer)] -> CurrencySymbol -> Bool
  isNFTBurned fVal symbol = (== 1) . length . filter (\(cs,_,nb) -> cs == symbol && nb == -1) fVal
```

Finally, we suggest to use use `haskell-language-server` and apply suggestions from the tool whenever applicable. This is of great help to get used to Haskell common programming style.

2.3 Conclusion

This report outlines the 16 concerns that we have gathered while inspecting the design and code of Cerra P2P Lending, pertaining to the code contained in the files listed in Table 2.1. As stated in Chapter 1, Tweag does not recommend for nor against the use of any work referenced in this report. Nevertheless, the existence of *critical* severity concerns is a warning sign.