TrueFi
Smart Contracts Audit

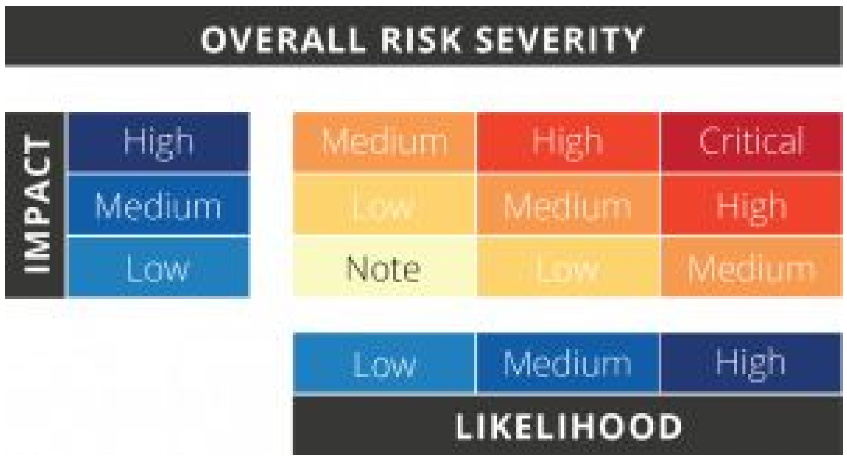
INTRO

The Trust Token team is working on the under-collateralized lending system powered by the TRU token incentives. They asked me to audit the protocol that includes tokenized loans, a lending pool and a smart-contract powered rating agency.

The code which is audited is located in the open-source github repository trusttoken/smart-contracts/ and the scope is limited to contracts located in the contracts/truefi folder.

The version of the code that is being reviewed was published with the commit `f34c5737cf80c84362217e65b70297cdcb5a485c`.

All of the issues are classified using the popular OWASP risk rating model. It estimates the severity taking into account both the likelihood of occurrence and the impact of consequences.



OVERALL RISK SEVERITY

IMPACT			
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Note	Low	Medium
	Low	Medium	High
	LIKELIHOOD		

SUMMARY

The development team demonstrated solid engineering skills to implement a very ambitious project combining multiple DeFi patterns: lending pool, farming and prediction markets. The design is clean and the responsibilities are well divided between the smart contracts. The code is readable, functions are well-scoped and the documentation is mostly sufficient to understand the code.

.

There have been **no critical** and 5 **high** severity issues found. There are also 11 medium and 15 low severity issues that in the edge cases could have severe side-effects. The main vulnerabilities include **frozen funds in edge cases, abusing incentives with flash-loan funding and front-running, calculation errors and lack of proper argument verification.**

ISSUES FOUND

Frozen late payments

Severity: **HIGH** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `LoanToken` allows a loan to be repaid even after the `term` is over and the loan is closed. Therefore it's possible that all of the `LoanToken` supply is already burned and there are no more tokens to claim new funds, making the new repayment inaccessible.

CONSEQUENCES

If a user pays back a loan in more than a single payment there is a risk that the borrower uses all of the tokens to claim the funds returned in the first repayment. Any subsequent late-payment will be forever locked in the contract and won't be accessible either to a lender or borrower.

RECOMMENDATIONS

I suggest preventing the above scenario, either by blocking payments if the token supply is zero. I also recommend considering an admin-controlled token reclaim function that could be used in edge-cases, like dormant loans untouched for a long period after being closed. This could be useful at the early stage with inexperienced users and untested front-end.

Lender risk calculation skips important factors

Severity: **HIGH** (Impact: HIGH, Likelihood: HIGH)

PROBLEM

The `votesTresholdReached` and `loanIsCredible` functions from `TrueLender` contract calculate the risk of funding a loan based on the total value of votes staked on predicting loan repayment. However, the estimate doesn't take into account `lossFactor` and `burnFactor` defined in the `TrueRatingAgency`.

CONSEQUENCES

The risk of staking Tru Tokens to vote depends on the potential loss that may occur if the prediction was wrong. If the `lossFactor` is near zero there is a very low risk for the voters in case default and equally low potential for gains on the successful prediction. In that scenario, the calculation of `loanIsCredible` won't be accurate as the computed risk aversion (*riskAversion = 10,000 => expected value of 1*) won't match the real risk exposure.

RECOMMENDATIONS

I suggest incorporating the values of `lossFactor` and `burnFactor` in the formulas used to quantify the risk for lenders.

Draining rewards by fake funding with flash-loans

Severity: **HIGH** (Impact: HIGH, Likelihood: MEDIUM)

PROBLEM

The `LoanToken` could be successfully processed with funds borrowed from a flash-loan. Both funding and withdrawal may happen in a single transaction allowing anyone to take a flash-loan, fund, withdraw to special-purpose smart-contract and return the flash-borrowed funds.

CONSEQUENCES

A malicious actor may abuse the flash-loan funding mechanism to create multiple high-value loans that could be eligible for rewards proportional to the loan size. That could lead to draining the participation reward pool of the `RatingAgency` contract.

RECOMMENDATIONS

I suggest preventing the possibility of funding a `LoanToken` and withdrawing the assets in a single transaction. It’s possible by introducing a minimum waiting period (at least one block) after which the deposited funds could be accessed by the borrower.

Possibility of TUSD / yUSD arbitrage

Severity: **HIGH** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `TrueFiPool` contract allows users to join and exit the pool in a single transaction. The pool assumes a $1.0 price per TUSD and a curve virtual price for yUSD. When these values differ from a spot market price there is an opportunity to exchange TUSD for yUSD by joining and exiting the pool without suffering any slippage.

CONSEQUENCES

The pool should serve the purpose of a long-term investment vehicle and shouldn't be abused to quickly exchange one token into another. Any profit realised by an arbitrage trader could mean a potential loss for all of the pool depositors and should be avoided.

Moreover, the ability to process joining and exiting in a single transaction brings a risk of using flash-loan funds to execute the trade and perform the large-scale arbitrage.

RECOMMENDATIONS

I recommend reducing the risk of arbitrage and preventing the possibility of flash-loan funding by adding an obligatory waiting period between joining the pool and requesting a withdrawal.

Lost farming rewards for the pre-staking period

Severity: **MEDIUM** (Impact: MEDIUM, Likelihood: MEDIUM)

PROBLEM

The `update` modifier from the `TrueFarm` contract distributes the funds before recording the stake. However, for the period before the first stake, funds are lost because the transferred value is not registered as a claimable reward on user accounts.

CONSEQUENCES

The only way to extract funds from the `TrueFarm` contract is by claiming rewards. Therefore, all the funds deposited from the distributor should be fully allocated to the staking users as they become lost otherwise.

RECOMMENDATIONS

I strongly recommend skipping the distribution and exiting the `update` function early if the `totalStaked == 0` or starting the distribution period only after the first stake is being made.

Votes escaping by front-running funding / rating reentrance

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `withdraw` method from the `TrueRatingAgency` contract allows removal of stake before the loan is funded. However, it's possible to front-run the funding transaction and remove the votes at the very last moment.

CONSEQUENCES

Voting plays an important role in signalling if a loan is likely to be repaid and involves staking tokens which could be lost if the prediction is incorrect. However, allowing users to withdraw the stake before funding may remove the risk factor and skew incentives.

This could also enable borrowers to stake large amounts of funds for voting for a deliberately defaulted loan and avoid the punishment removing votes just before the funding happens.

Although the `TrueLender` checks the votes quorum and period while funding in one transaction, it does so before calling the `borrow` method that invokes a 3rd party Curve contract which may possibly allow entering `TrueRatingAgency`.

RECOMMENDATIONS

I recommend adding a cool-down / lock period during which votes cannot be withdrawn that could mitigate the problem for human lenders.

For the smart-contract based `TrueLender` I recommend calling all of the methods invoking 3rd party contracts before checking rating conditions. It's a golden rule of security never to release control to external code between conditions checking and executing an action.

Uncontrolled curve liquidity removal

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `borrow` method `TrueFiPool` contract withdraws liquidity from the Curve protocol by calling `remove_liquidity_one_coin`. This operation is prone to price manipulation, front-running and slippage, therefore it should be executed with extreme care.

CONSEQUENCES

The borrowing operation is initiated by one of the `allowedBorrowers` of the `TrueLender` contract. If this privilege is widely distributed to community members we may assume that one of them could act in a malicious way. One of the possible attack vectors is to take a flash-loan to manipulate the current TUSD curve price, allow the withdrawal to happen at an unfavourable price and exchange the funds back, skimming the profit.

Even if the `allowedBorrowers` are considered to be fully trusted and couldn't act in a malicious way, there is still a risk of exchanging a very large amount of funds in a single transaction. Such an attempt could be easily front-ran or incur a significant slippage.

RECOMMENDATIONS

Because of the risk mentioned above, I recommend considering a two-phase borrow process, when a lender signals that the loan is ready to be borrowed but the necessary funds could be accumulated by the TruefiPool owner who decides on the volumes and timing of curve integrations.

Inconsistent bounty calculation

Severity: **MEDIUM** (Impact: MEDIUM, Likelihood: LOW)

PROBLEM

A reward / penalty for making a prediction is calculated in two places in the `TrueRatingAgency` contract. The formulas used in the `withdraw` and the `bounty` method differ slightly in terms of rounding where the first one executes two divisions by `10000` while the second one performs only a single division by the `10000 ** 2` factor.

CONSEQUENCES

The staking bounty for correct prediction is taken from the losses of users who predicted the wrong loan outcome. If the values mismatch there is a risk that the contract may break due to the deficit on contract or part of the funds may be left on balance and become inaccessible.

RECOMMENDATIONS

I recommend having a single formula for bounty calculation and use it across the contract to avoid values mismatch and rewards/punishment imbalance.

Unsafe farming with inflationary/deflationary tokens

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `TrueFarm` contract takes a snapshot of user balance at the moment of staking and uses it to update its internal accounting. However, the accounts are not synced again meaning it could keep stale values for deflationary or inflationary tokens.

CONSEQUENCES

Variable-balance tokens are getting more and more popular. Some of them could automatically increase user balance like ones from the Aave lending platform. Others can readjust account holding based on the current price, like Ampleforth or Yam. These tokens require special treatment and can cause funds to be lost or frozen while used with the current version of the `TrueFarm` contract.

RECOMMENDATIONS

I suggest documenting explicitly that the solution is not compatible with variable-balance tokens. However, if there is a business need to integrate with this new type of assets funds should be deposited to a special-purpose user account capable of syncing the balance.

Insufficient funds on distributor may block funds on farm

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `exit` function from `TrueFarm` contract always executes the code from the `update` modifier before releasing the funds. This code is dependent on the completion of the call to the `trueDistributor` which may break if there are not enough funds deposited.

CONSEQUENCES

One of the most important security guarantees required by the farmers is the ability to maintain control of the funds deposited. However, releasing the funds is dependent on the behaviour of an external contract and the liquidity provided could be easily frozen if the owner calls the `empty` method from the `LinearTrueDistributor`.

RECOMMENDATIONS

I suggest offering farmers a strong guarantee of control over their funds either by allowing the `distribute` method to execute gracefully despite lack of funds or by adding an `escape` method allowing users to withdraw funds without going through rewards claiming.

It should be possible to view currently accumulated rewards

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

A user delegating liquidity to the `TrueFarm` contract cannot check the accumulated rewards before claiming the funds and paying gas for the transaction.

CONSEQUENCES

The ability to check accumulated reward in real-time is an important feature enabling integration with various farming aggregators and cash-flow engines. Users should be able to check their current rewards balance using a view-only function that doesn't require gas payments.

RECOMMENDATIONS

I suggest refactoring the code of the `update` modifier and extracting the code responsible for rewards calculation to a new, view-only method.

Hijacking loan allowance

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `repay` function from the `LoanToken` contract allows the caller to initiate repayment from any user account that has previously made an allowance to the contract.

CONSEQUENCES

One of the potential exploit scenarios is using the funder allowance to repay a loan. Let's assume that a funder decided to choose unlimited approval, which is a common practice of many clients. Such an allowance could be maliciously used to repay the loan without any awareness of the owner.

RECOMMENDATIONS

If the protocol requires 3rd party initiated payments, I strongly recommend double-checking the infinite approvals or programmatically blocking repayments from the funder account.

Frequent distributions may block funds

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `distribute` method updates the `LinearTrueDistributor` state even if the calculated amount is fractional and no funds are sent.

CONSEQUENCES

Updating the `lastDistribution` variable without distributing any token may block the fractional funds and corrupt distribution schedule. The `distribute` method is open to anyone, so a malicious actor may successfully block a distribution providing the token precision or distributed amount is low enough.

RECOMMENDATIONS

I recommend updating the `lastDistribution` state variable only if the distribution happens, meaning the calculated `amount > 0`.

Censoring 3rd party loans

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

Any user may submit a loan to the `TrueRatingAgency` contract without the need to be a creator, an owner or a borrower. The loan could be immediately retracted blocking resubmission from the legitimate owner.

CONSEQUENCES

A malicious actor may prevent any loan from entering the `TrueRatingAgency` by submitting and retracting. The unauthorised censorship may prevent legitimate users from obtaining the ranking and limit their ability to raise funds.

RECOMMENDATIONS

I recommend adding a verification mechanism to the submission process requiring the loan creator to authorise submission.

Distributor may block claiming earned rewards on Rating

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `claim` function from `TrueRatingAgency` contract always executes the code from the `calculateTotalRewars` modifier before releasing the rewards. This code is dependent on the completion of the call to the `distributor` which may break if there are not enough funds deposited.

CONSEQUENCES

All of the past rewards earned by users must remain available to them. The lack of funds on the distributor account should not block access to the funds that have already been transferred to the `TrueRatingAgency`.

RECOMMENDATIONS

I suggest allowing users to claim past rewards without calling rewards recalculation, in case there are not enough funds on the distributor contract.

Arbitrary distributor could not be topped up

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `ArbitraryDistributor` specifies the maximum `amount` of tokens to be distributed during the initialization. That value could not be changed later, blocking the possibility of distributing more tokens.

CONSEQUENCES

When the `ArbitraryDistributor` is connected to the `TrueRatingAgency` it is not possible to quantify apriori the total amount of rewards as they depend on the total loan profit which is proportional to loan number, volume and interest rates. Therefore, it's possible that the rewards value will exceed the amount defined in the `ArbitraryDistributor` which could break the mechanism of rewards distribution.

RECOMMENDATIONS

I recommend allowing the ArbitraryDistributor to increase the amount and open it to distribute more tokens according to the needs.

Distribution contract should prevent rewards pre-mining

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

Setting the `distributionStart` parameter of the `initialize` function from the `LinearTrueDistributor` contract to a past value enables the first beneficiary to claim an unjustified portion of rewards.

CONSEQUENCES

Having an unequal reward for the first caller of the `distribute` function could create a race condition between users paying extreme gas fees to claim the early rewards.

RECOMMENDATIONS

I recommend verifying the `distributionStart` parameter by adding a check `require(distributionStart >= block.timestamp)` in the `initialize` function.

Repay should protect against overpayments

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `repay` function of the `LoanToken` contract allows any amount to be paid back regardless of the actual user liability.

CONSEQUENCES

If a user miscalculates the return amount or makes an input error the over payments are transferred in full and couldn't be returned back to the borrower. This allows the lender to unjustly benefit from the borrower's mistakes. The protocol should programmatically disable such scenarios, especially during the launch phase, when the users are not fully familiar with the interface.

RECOMMENDATIONS

I suggest validating the repayment amount to check if it doesn't exceed the full debt value.

RatingAgency should validate loss and burn factors

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `setLossFactor` and the `setBurnFactor` method of the `TrueRatingAgency` contract don't validate the input parameters.

CONSEQUENCES

A user who staked tokens and analysed the risk may expect certain loss based on the loss and burn factors. However, the owner of the `TrueRatingAgency` can change these parameters even during the phase when the user cannot react and withdraw staked tokens. Therefore it's important to move the values within a certain range to avoid drastically changing the rules during the game.

There is also a possibility that the values are set above a 100% which can break the logic of rewards/punishment calculation and seize unjustified amount of funds.

RECOMMENDATIONS

I recommend verifying both the `setLossFactor` and `setBurnFactor` not to exceed 100%. Moreover, I suggest coding the maximum allowed value to limit users' risk level.

It should not be possible to create a zero-term loan

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `LoanToken` constructor allows users to create a loan with a zero-valued `term` parameter. Such a loan could be correctly funded, withdrawn and closed in a single transaction.

CONSEQUENCES

The ability to fund and settle a loan within a single transaction could be abused to improve repayment statistics or claim unjust incentives.

The zero-term loan could also be very dangerous if connected to a time-based pro-rata reward mechanism. Because of the zero duration the reward calculation logic, like the one in the `claim` function from the `RatingAgency` contract, could be broken due to the division by zero.

RECOMMENDATIONS

I suggest validating the input arguments by adding a check: `require(term > 0).`

It should not be possible to create empty loan

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `LoanToken` constructor allows users to create a loan with a zero `amount`. Such a loan could be correctly processed through all of the stages requiring no financial commitments and token transfers.

CONSEQUENCES

The ability to create loan entities without any need for funds to be processed could be exploited to spam the system and populate the network with hundreds of empty contracts. The hollow loans may pollute the TrueFi protocol making it hard to provide accurate activity data and user statistics.

The system should also flag that a zero-valued loan is not correct if the empty value was submitted due to a user error or a front-end issue.

RECOMMENDATIONS

I suggest validating the input arguments by adding a check: `require(amount > 0).`

Distributor should validate the trust token address

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `_trustToken` parameter of the `LinearTrueDistributor` contract is not validated. Therefore it is possible to pass a null value and break the contract functionality without any possibility of fixing the issue.

CONSEQUENCES

The major risk of setting an incorrect / null token address is blocking the possibility of releasing the funds. If the contract has been funded from the correct trust token address, the `distribute` and `empty` still refer to the wrong address, locking the funds.

RECOMMENDATIONS

I suggest validating the address by adding a check `require(trustToken != address(0)`. I also strongly recommend extending the `empty` function to accept a token address as one of the parameters to have full flexibility of reclaiming the funds in case of any mistake.

Error in value calculation for zero-supply loans

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `value` function of the `LoanToken` contract might throw a low-level calculation error when invoked for a zero-supply loan.

CONSEQUENCES

The `value` function should always return a loan value regardless of the current loan status and the balance being queried. If an unfunded or a fully-repaid loan is aggregated in a pool or an index with other loans such an error could prevent calculation of the aggregated value and break the logic of the client contract.

RECOMMENDATIONS

I strongly recommend having an edge-case check for the token supply and returning a zero value for the unfunded or fully reclaimed loan.

Initializing zero-duration distribution should be prohibited

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `initialize` function of the `LinearTrueDistributor` contract allows a zero value as a duration.

CONSEQUENCES

Setting the `duration` variable to zero will break the logic of the `distribute` function and block any fund from being distributed. As the variable could not be corrected in the future it is crucial to be verified.

RECOMMENDATIONS

I recommend verifying the `duration` parameter by adding a check `require(duration > 0)` in the `initialize` function.

Distribution should be paused until the farm is set

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `distribute` method, that could be invoked by anyone, could transfer tokens to a null address if the `farm` target is not set.

CONSEQUENCES

The `LinearTrueDistributor` could burn funds if a malicious user calls the `distribute` method before the `farm` variable is set. Although some of the token implementations may block sending to the `0x0` address the contract should not be active until it is properly initialized.

RECOMMENDATIONS

I recommend skipping the distribute function by verifying the farm variable is set and early returning from execution if it has not been initialized yet.

Distributor should be validated in TrueFarm

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `initialize` method of the `TrueFarm` doesn't perform the necessary checks on the distributor allowing a wrongly configured distributor to be used during farming.

CONSEQUENCES

Because the initialize method can be called only once it's extremely important to check and sanitize all of the arguments. The farming should not be connected to an uninitialized distributor or to a distributor that internally points to a wrong farm address.

RECOMMENDATIONS

I recommend verifying the `_trueDistributor` parameter by adding a check `require(_trueDistributor.farm == address(this))` in the `initialize` function.

TrueFarm should validate the staking token

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `initialize` method of the `TrueFarm` should check if a staking token is not null.

CONSEQUENCES

The initialize method can be called only once, therefore it's extremely important to check and sanitize all of the arguments. If the stalking token points to a null address the farm becomes unusable and funds released from the distributor are lost.

RECOMMENDATIONS

I recommend verifying the `_stakingToken` parameter by adding a check `require(address(_stakingToken) != address(0))` in the `initialize` function.

TrueRatingAgency should validate the distributor

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `initialize` method of the `TrueRatingAgency` doesn't perform any checks on the `_distributor` allowing a wrongly configured contract to be connected.

CONSEQUENCES

The initialize method can be called only once, so it's extremely important to check and sanitize all of the arguments. There is a risk that the rating contract is connected to a distributor pointing to a wrong address and the rewards won't be available. Such a problem could only be discovered at a later stage when rewards become available after funding a loan.

RECOMMENDATIONS

I recommend verifying the `_distributor` parameter by adding a check `require(_distributor.beneficiary == address(this))` in the `initialize` function.

Unbounded loops in TrueLender

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `TrueLender` contract contains three unbounded loops in the `value`, `distribute` and `reclaim` functions.

CONSEQUENCES

Due to the computation model of the EVM having unbounded loops is considered to be an unsafe practice as running through all of the elements may consume the full block gas amount. If the issue is overlooked and the protocol grows, keeping an increasing number of loans, it could be broken.

Moreover, once the contract becomes broken there is no way to fix the issue as removing a loan also requires looping through all of the existing entries.

RECOMMENDATIONS

I strongly recommend adding a constant defining a maximum safe number of loans before deploying the code to production.

Lender should check currencies consistency

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `TrueLender` contract should verify that the currency of the `LoanToken` matches its own currency.

CONSEQUENCES

Both the `TrueLender` and `LoanToken` can specify their currencies that don't need to point to the same token contract. The inconsistency should be caught as early as possible and correctly reported to users.

A malicious player may abuse this inconsistency to drain funds from the pool. It may submit a loan denominated in a valueless token, deposit that token directly on the lender contract and use the loan to borrow funds from the pool. Currently, this scenario is mitigated by the `RatingAgency` verification, but it could become dangerous if rating requirements are softened.

RECOMMENDATIONS

I recommend verifying token consistency by adding an extra check in the `fund` method: `require(loanToken.currencyToken == currencyToken).`

Indefinite funds approval from the pool to 3rd party contract

Severity: **LOW** (Impact: HIGH, Likelihood: LOW)

PROBLEM

The `TrueFiPool` grants unlimited approval to the `curvePool` contract delegating the full control of the depositors' funds.

CONSEQUENCES

The indefinite approval is considered a risky pattern as the funds could be drained if the implementation of the target contract changes. There is also a risk that the target curve contract becomes compromised and the hacker will use it as a proxy to remove funds from the `TrueFiPool.`

RECOMMENDATIONS

I recommend granting minimum possible approvals to the external contracts each time there is a need for a delegated transfer.

Notes

Issues with a very low impact

WRONG DESCRIPTION OF THE REPAID METHOD

The `repaid` method from the `LoanToken` contract is described to return "*Boolean representing whether the loan has been repaid or not*". However, the method returns the repaid amount as an `uint256` value.

WRONG DESCRIPTION OF THE INTEREST METHOD

The `interest` method from the `LoanToken` contract is described to return "*Amount of interest paid for _amount*". However, the method returns the full loan amount plus interests.

DUPLICATED INTEREST CALCULATION CODE

The interest calculation code is duplicated in the `LoanToken`. Both the `interest` and the `value` method implements the same logic. I recommend avoiding code duplication as it increases the maintenance cost and brings the risk of an error caused by updating the code only in a single location.

LOAN REPAYMENT SHOULD BE SIGNALLED WITH AN EVENT

All of the important actions that change the `LoanToken` state emits events apart from the `repay` method. I recommend emitting an event when the loan is repaid to facilitate loan tracking and simplify front-end integration.

BORROWER FEE SHOULD BE DEFINED AS A CONSTANT

The `borrowerFee` variable used in the `LoanToken` contract cannot change its value. I recommend defining it as a constant parameter. This will not only improve code readability but also reduce gas costs.

THE REDEMPTION AMOUNT SHOULD BE VALIDATED

The `redeem` method of the `LoanToken` contract should throw an error if the requested `_amount` value exceeds user balance. This will not only improve debugging and user messaging but it may also reduce the gas costs of incorrect invocations.

THE LOAN SHOULD NOT BE FUNDED FROM THE BORROWER ACCOUNT

Lender and borrower are two opposite parties involved in loan processing and using the same account to perform both roles is a clear indication that a loan could have a fictional status and is created either to manipulate statistics or abuse the incentive system.

UNUSED ARGUMENT IN THE DISTRIBUTE METHOD

The `distribute` method of the `LinearTrueDistributor` contract has an unnamed address input parameter. The parameter is never used in the code. This could be misleading as it may suggest that the address passed is the distribution target while the real destination is defined by the `farm` variable.

LENDER SHOULD VERIFY LIMIT ARGUMENTS

The loan acceptance requirements set in the `TrueLender` contract by calling one of the `setSizeLimits`, `setTermLimits`, `setApyLimits`, `setVotingPeriod`, `setParticipationFactor` and `setRiskAversion` are not properly sanitized. I recommend checking if the values are greater than zero and below the max possible value for all of the parameters.

THE LOAN TOKEN VALIDATION IN LENDER FUNDING IS NOT EFFECTIVE

The `fund` method from the `TrueLender` checks if a contract returns true from the `isLoanToken` method. However, this condition could be easily met by any contract created by any malicious party. I do recommend using a solution that is currently implemented in the `TruRatingAgency` and based on `LoanFactory`. That solution will also mitigate the problem with currency consistencies check reported as a low severity issue above.

USE NAMED CONSTANTS INSTEAD OF MAGIC VALUES

In many places across the contracts, there are occurrences of the `10000` raw integer value. They act as a 100% literal for calculations. Such numbers are often referred to as "magic values" and should be replaced by constant declaration to increase readability and eliminate the risk of forgetting to update one of them during the refactoring.

TYPOS IN THE RATING AGENCY CONTRACT

There are a few typos in the comments of the `TrueRatingAgency` contract. Where it says: "*muliple*" it should say "multiple". Moreover, when it says "*borrwers*" (twice) it should say "borrowers".

POOL REPAYMENTS SHOULD BE RESTRICTED TO LENDER

Anyone is allowed to send a minimal amount of funds to trigger the `Repaid` event in the `TrueRatingAgency` contract. This could pollute events space and make it harder to monitor paid and active loans. I recommend restricting access to this function only to the `lender` account.

TOKEN PRECISION RATIO SHOULDN'T BE HARDCODED

The `toTrustToken` method from the `TrueRatingAgency` contract uses a hardcoded constant `TOKEN_PRECISION_DIFFERENCE` to translate values from one token to another. That limits the possibility of using the code with other tokens and can cause hard to find errors if new types of assets are used.

Jakub Wojciechowski
Smart contract auditor