Audit Report

Umee Leverage Module

Delivered: June 9, 2022

Prepared for Umee by Runtime Verification, Inc.

Summary

Runtime Verification, Inc. has audited the source code for the Umee leverage module. The review was conducted from 2022-04-18 to 2022-05-20.

Umee engaged Runtime Verification in checking the security of their Cosmos chain, specifically the Leverage Module, which implements a decentralized money market, allowing lending and collateralized borrowing. All prices are normalized against USD and supplied by an external oracle system.

Scope

The audited code is:

- The `x/leverage` module, written in Golang on top of Cosmos SDK.

The audit has focused on the above module, and has assumed correctness of the libraries and external systems it makes use of.

The review encompassed 1 public code repository.

- The code was frozen for review at commit: `28028987ea5d3af53be339df143784f25a070b7f`.

The review is limited in scope to consider only module code. Non-chain and client-side portions of the codebase are *not* in the scope of this engagement.

Assumptions

The audit is based on the following assumptions and trust model.

1. We assume that the Umee chain operates normally at all times, and that the validator set remains honest to >⅔.
2. We assume that the oracle price feed is reliable and not manipulable by any one validator (the oracle module is out of scope for this audit, but has been previously audited).
3. We assume that governance is honest and competent, and sets reasonable protocol parameters. That is to say, it will not sabotage the protocol only for the sake of sabotage.

Note that assumption 3 roughly assumes honesty and competence. However, we will rely less on competence, and point out wherever possible how the module could better ensure that unintended mistakes cannot happen.

We do not assume full trust-worthiness of governance, and do **not** for example trust it with the ability to block certain users or take control of funds that the user did not intend to allow (roughly: theft). That is to say, we assume that governance would potentially sabotage the protocol if there was a clear monetary gain in doing so.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the module, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to known security issues and attack vectors. Thirdly, we discussed the most catastrophic outcomes with the team, and reasoned backwards from their places in the code to ensure that they are not reachable in any unintended way. Finally, we regularly participated in meetings with Umee and offered our feedback during ongoing design discussions, and suggested development practices as well as design improvements.

This report describes the **intended** behavior and invariants of the contracts under review, and then outlines issues we have found, both in the intended behavior and in the ways the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter. Finally, we also give an overview of the important security properties we proved during the course of the review.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Umee leverage: Module Description and Invariants

The Umee chain is a Cosmos zone, built on the Cosmos SDK. As such it extends the base implementation from the Cosmos SDK with modules that perform specific functions. The role of modules is similar to that of contracts on smart contract chains, with some differences. For example, it is possible to trigger some interaction with the modules at the end of each block, without any user performing a transaction.

This section describes the module at a high-level, and which invariants of its state we expect it to always respect at the end of a transaction or a block.

Leverage Module | Parameters and Variables

The leverage module hosts the key financial logic of Umee protocol. To facilitate the financial activities (e.g., lending, borrowing, liquidating etc.) of the protocol, the module keeps track of the important statistics as state variables such as the interest scalar, the borrow amount etc. It also replies on the $x/bank$ module from cosmos SDK to keep tracking of amount balances and minting uTokens and on the $x/oracle$ module to provide the latest token price in USD.

We will introduce these state variables in the following along with an abstract model[1] of the protocol to facilitate reasoning of the finance logic. For example, we represent the balance of a token $denom$ of a user[2] usr kept in x/bank module[3] using $bank[usr][denom]$ and the price of a token $denom$ using $TokenPrice[denom]$.

RegisteredToken is a registry governed by validators. It kept a list of tokens that are accepted in Umee's lending protocol. This state variable has the structure that maps a base token to a Token structure (listed below), where these token parameters are used in main functionalities like liquidation. We will use $RegisteredToken[denom].LiquidationThreshold$ to access these parameters within this document.

```
type Token struct {
    BaseDenom            string
    ReserveFactor        sdk.Dec
    CollateralWeight     sdk.Dec
    LiquidationThreshold sdk.Dec
    BaseBorrowRate       sdk.Dec
```

[1] The abstract model uses an array style representation which does not necessarily reflect the actual implemented data structure.
[2] A COSMOS chain compatible address.
[3] belongs to cosmos SDK and assumed as a trusted module in this audit.

```
    KinkBorrowRate       sdk.Dec
    MaxBorrowRate        sdk.Dec
    KinkUtilizationRate  sdk.Dec
    LiquidationIncentive sdk.Dec
    SymbolDenom          string
    Exponent             uint32
}
```

Here,

$$TokenPrice(BaseDenom) = 10^{Exponent} * TokenPrice(SymbolDenom)$$

AdjustedBorrow records all the borrows for each base token. The variable will be updated during a block. We would access it, in this document, in an abstracted way: $AdjustedBorrow[borrower][denom]$, referring to the amount of the $denom$ borrowed by the $borrower$. For example, if the borrower, b, successfully borrowed the denom d of an amount amt, the AdjustedBorrow variable is updated[4] as

$$AdjustedBorrow[b][d] \mathrel{+}= amt\ /\ InterestScalar[d]$$

AdjustedTotalBorrow is the variable tracking the total borrowed amount of a denom. This variable is updated passively with each AdjustedBorrow update. Similarly, we represent the total adjusted borrow amount for the denom, $denom$ usingThe relation between these two variables are

$$AdjustedTotalBorrow = \Sigma_{b \in B} AdjustedBorrow[b]$$

where B is the set of all borrowers.

InterestScalar is the variable updated at the end of each block. It records the latest interest rate for borrowing each base token. Thus, to calculate the total borrowed amount for a denom, d, we have

$$TotalBorrow[d] = AdjustedTotalBorrow[d] * InterestScalar[d]$$

Similarly, the total borrowed amount of specific borrowers are calculated likewise.

UTokenSupply is the variable tracking the total supply of each uToken. We refer to the total supply of a uToken u as $UTokenSupply[u]$.

CollateralSetting records the base tokens that are collateralized. In the case that a borrower b collateralized the token d, we represent it as $CollateralSetting[b][d] = true$.

[4] The variable InterestScalar is introduced in the following.

CollateralAmount records the total collateralized amount of a base token. This variable to be used to calculate the borrow limit, liquidation reward etc.. In our abstract model, we use $CollateralAmount[b][d]$ to represent the collateralized amount of the base token, d of the borrower, b.

ReserveAmount keeps track of the amount that is reserved for paying bad debt of a base token. We represent it using $ReservedAmount[denom]$. A portion, depending on the reserve factor of the base token, of the accrued interest at the end of the block is reserved.

LastInterestTime is updated at the end of a block when accrued interest is calculated. The current blocktime will be the new value.

BadDebt keeps track of borrowed assets of borrowers which are labeled as bad debt. It is set to be true when the borrower's collateral is drained in a liquidation event, we represent it in this document as $BadDebt[borrower][denom] = true$. The variable will be reviewed at the end of a block when the sweep bad debt event occurs.

There also four parameters set at the module level, which are

OracleRewardFactor with a default value 0.01 which decides the portion of the accrued interest that will go to the oracle module account.

MinimumCloseFactor decides the close factor just above a borrower’s liquidation limit. Along with the following two parameters, are used in a liquidation event to decide the eligible liquidation amount.

CompleteLiquidationThreshold decides when the borrower’s borrowing can be fully liquidated in a single liquidation event.

SmallLiquidationSize allows full liquidation when the borrower's borrowed value is less than it.

Leverage Module | Rates

There are multiple rates involved in the financial activities, such as exchange rate from base token to uToken, borrow rate etc.. We summarize their calculation logic (according to implementation) in the following.

Before deriving any functions, we’d summarize a few common derived variables:

The total supply of a base token $denom$ is calculate as

$$TokenSupply[denom]$$

$$= ModuleBalance[denom] + TotalBorrowed[denom] - reserved[denom]$$

where

$$ModuleBalance[denom] = bank[leverage][denom]$$

$$TotalBorrowed[denom] = \lceil AdjustedTotalBorrowed[denom] \times InterestScalar[denom] \rceil^{5}$$

uToken Exchange rate

The token $denom$ when lended to the leverage module, will be converted to its uToken form $u/denom$ using the following exchange rate[6] :

Case 1: when $uTokenSupply[u/denom] \leq 0$,

$$uExchangeRate[denom] = 1.0$$

Case 2: when $uTokenSupply[u/denom] > 0$,

$$uExchangeRate[denom] = \frac{tokenSupply[denom]}{uTokenSupply[u/denom]}$$

Where

The exchange rate was calculated in the function DeriveExchangeRate(ctx, denom) in x/leverage/keeper.go as follows:

```go
// DeriveExchangeRate calculated the token:uToken exchange rate of a base token
denom.
func (k Keeper) DeriveExchangeRate(ctx sdk.Context, denom string) sdk.Dec {
  // uToken exchange rate is equal to the token supply (including borrowed
  // tokens yet to be repaid and excluding tokens reserved) divided by total
  // uTokens in circulation.

  // Get relevant quantities
  moduleBalance := k.ModuleBalance(ctx, denom).ToDec()
  reserveAmount := k.GetReserveAmount(ctx, denom).ToDec()
  totalBorrowed := k.getAdjustedTotalBorrowed(ctx,
denom).Mul(k.getInterestScalar(ctx, denom))
  uTokenSupply := k.GetUTokenSupply(ctx, k.FromTokenToUTokenDenom(ctx,
denom)).Amount

  // Derive effective token supply
```

[5] ⌈⌉ is a ceiling function, where in Umee's context, ⌈d⌉=d.ceil().TruncateInt()
[6] In the document, without causing ambiguity, "exchange rate" refers to token to uToken exchange rate.

```
  tokenSupply := moduleBalance.Add(totalBorrowed).Sub(reserveAmount)

  // Handle uToken supply == 0 case
  if !uTokenSupply.IsPositive() {
     return sdk.OneDec()
  }

  // Derive exchange rate
  return tokenSupply.QuoInt(uTokenSupply)
}
```

Utilization Rate

The utilization rate refers to the percentage of the borrowed amount of a token *denom*. The logic of the calculation is summarized as follows:

Case 1: when $TotalBorrowed > TokenSupply$,

$$UtilizationRate[denom] = 1.0$$

Case 2: when $TotalBorrowed \leq TokenSupply$

$$UtilizationRate[denom] = \frac{TotalBorrowed[denom]}{TokenSupply[denom]}$$

The utilization rate is calculated in the function *DeriveBorrowUtilization*(*ctx*, *denom*) of x/leverage/borrows.go.

```
// DeriveBorrowUtilization derives the current borrow utilization of a token denom.
func (k Keeper) DeriveBorrowUtilization(ctx sdk.Context, denom string) sdk.Dec {
  // Borrow utilization is equal to total borrows divided by the token supply
  // (including borrowed tokens yet to be repaid and excluding tokens reserved).
  moduleBalance := k.ModuleBalance(ctx, denom).ToDec()
  reserveAmount := k.GetReserveAmount(ctx, denom).ToDec()
  totalBorrowed := k.GetTotalBorrowed(ctx, denom).Amount.ToDec()

  // Derive effective token supply
  tokenSupply := totalBorrowed.Add(moduleBalance).Sub(reserveAmount)

  // This edge case can be safely interpreted as 100% utilization.
  if totalBorrowed.GTE(tokenSupply) {
     return sdk.OneDec()
  }

  // Derive borrow utilization
  return totalBorrowed.Quo(tokenSupply)
}
```

Borrow Rate

For a particular base token *denom*, at a utilization rate u, its borrow rate r (yearly rate) is calculated as

$$r = [u < U_{kink}] * (\frac{R_{kink} - R_{base}}{U_{kink}} * u + R_{base})$$
$$+ [u \geq U_{kink}] * (\frac{R_{max} - R_{kink}}{1 - U_{kink}} * (u - U_{kink}) + R_{kink})$$

Where U_{kink}, R_{base}, R_{kink} are parameters of the registered token *denom*.

Interest Scalar

Interest scalar is updated at the end of each block. It is essential to normalize the borrowings that occurred at different times and accumulate the interest of all borrowings. The following code implements the logic of updating interest scalar.

```
   // iterate over all accepted token denominations
   for _, token := range tokens {
       // interest is accrued by multiplying each denom's Interest Scalar by the
       // quantity (borrowAPY * yearsElapsed) + 1
       scalar := k.getInterestScalar(ctx, token.BaseDenom)
       increase := k.DeriveBorrowAPY(ctx, token.BaseDenom).Mul(yearsElapsed)
       if err := k.setInterestScalar(ctx, token.BaseDenom,
scalar.Mul(increase.Add(sdk.OneDec()))); err != nil {
           return err
       }
```

Assume at k^{th} block (k > 1), for a particular denom d, the borrowings yields a new interest in this block of amount, $increase[d]_k$

$$increase[d]_k = r[d]_k * \frac{blocktime_k - blocktime_{k-1}}{secondPerYear}$$

Where $r[d]_k$ is the latest borrow rate for denom d at the end of this block.

Thus, the interest scalar of denom d at this block should be

$$InterestScalar[d]_k = InterestScalar[d]_{k-1} * (1 + increase[d]_k)$$
$$= Scalar_{genesis} * \prod_{i=1}^{k} (r[d]_i * \frac{blocktime_i - blocktime_{i-1}}{secondPerYear} + 1)$$

assuming its interest scalar at the genesis state is $Scalar_{genesis}$.

Lending Interest

For a base token $denom$, at its utilization rate u and borrow rate r, the lending interest rate l for this denom is derived as

$$l = r * u * (1.0 - oracleRewardFactor - RegisteredToken[denom].ReserveFactor)$$

Close Factor

This ratio parameter decides the fraction of a borrower's total borrowed value that can be liquidated in a single liquidation event. The calculation of this parameter is implemented as function $LiquidationParams()$ in x/leverage/keeper/keeper.go.

For the denom (i.e., desiredRepay.denom) that is being borrowed by $borrower$,

Case 1: when

$$borrowValue > SmallLiquidationSize \wedge LiquidationLimit > 0 \wedge CompleteLiquidationThreshold \neq 0$$

$$closeFactor = (\frac{1.0 - MinimumCloseFactor}{CompleteLiquidationThreshold} * (\frac{borrowValue}{LiquidationLimit} - 1.0) + MinimumCloseFactor)$$

Case 2: when

$$LiquidationLimit = 0 \vee borrowValue \leq SmallLiquidationSize \vee CompleteLiquidationThreshold$$

$$closeFactor = 1.0$$

If the above calculated $closeFactor \geq 1.0$, $closeFactor$ will be set to 1.0; however, if $closeFactor < 0$, $closeFactor$ is set to 0.0.

Reward Rate

In the liquidation event, if it is successful, the liquidator will receive a reward by paying back an eligible borrow. In such a case, the actual reward ratio, a.k.a., the amount received in the expected reward token $denom_{rw}$ versus the amount of repaid token $denom_{rp}$, is a constant value in a block,

$$\frac{tokenPrice[denom_{rp}]}{tokenPrice[denom_{rw}]} * (1.0 + RegisteredToken[denom_{rw}].LiquidationIncentive).$$

Furthermore, the liquidator submits a liquidation request with an expected reward. The function signature which defines this liquidation functionality is defined as

```
func (k Keeper) LiquidateBorrow(
   ctx sdk.Context, liquidatorAddr, borrowerAddr sdk.AccAddress, desiredRepayment,
desiredReward sdk.Coin,
) (sdk.Int, sdk.Int, error)
```

Where the desiredReward could be base tokens or uTokens. Thus, we have the user specified minimum ratio as

$$minimumRewardRatio = \frac{desiredReward.Amount}{desiredRepayment.Amount}.$$

While the actual reward ratio is calculated in the following codes (lines 411, 461-502 in x/leverage/keeper/keeper.go) in the current implementation. We break the code into blocks to facilitate the abstraction of our mathematical model.

```
   // get borrower uToken balances, for all uToken denoms enabled as collateral
   collateral := k.GetBorrowerCollateral(ctx, borrowerAddr)
```

$$collateral = CollateralAmount[borrowerAddr]$$

```
// Given repay denom and amount, use oracle to find equivalent amount of
// rewardDenom's base asset.
baseReward, err := k.EquivalentTokenValue(ctx, repayment, baseRewardDenom)
if err != nil {
  return sdk.ZeroInt(), sdk.ZeroInt(), err
}
```

we know $baseRewardDenom = desiredReward.Denom$

assume $rp_0 = repayment.Amount, rw_0 = baseReward.Amount$ both of type $sdk.Coin$

if $err = nil$, we have

$$rw_0 = \lfloor rp_0 * \frac{tokenPrice(rp.Denom)}{tokenPrice(rw.Denom)} \rfloor$$

```
// convert reward tokens back to uTokens
reward, err := k.ExchangeToken(ctx, baseReward)
if err != nil {
  return sdk.ZeroInt(), sdk.ZeroInt(), err
}
```

let $urw_0 = reward.Amount, uex = \frac{tokenSupply\ (rw.denom)}{uTokenSupply(u/rw.denom)}$,

if $err = nil$, we have

$$urw_0 = \lfloor \frac{rw_0}{uex} \rfloor$$

```
// apply liquidation incentive
reward.Amount =
reward.Amount.ToDec().Mul(sdk.OneDec().Add(liquidationIncentive)).TruncateInt()
```

let $i = liquidationIncentive$, we know

$$i = RegisteredToken[rw.Denom].LiquidationIncentive$$

$$urw_1 = urw_0 * (1 + i)$$

```
// reward amount cannot exceed available collateral
if reward.Amount.GT(collateral.AmountOf(reward.Denom)) {
  // reduce repayment.Amount to the maximum value permitted by the available
collateral reward
  repayment.Amount =
repayment.Amount.Mul(collateral.AmountOf(reward.Denom)).Quo(reward.Amount)
  // use all collateral of reward denom
  reward.Amount = collateral.AmountOf(reward.Denom)
}
```

if $urw_1 > collateral[rw.Denom]$, we have

$$urw_{if} = collateral[rw.Denom]$$

$$rp_{if} = rp_0 * \frac{urw_{if}}{urw_1}$$

```
// final check for invalid liquidation (negative/zero value after reductions above)
if !repayment.Amount.IsPositive() {
  return sdk.ZeroInt(), sdk.ZeroInt(), sdkerrors.Wrap(types.ErrInvalidAsset,
repayment.String())
}
```

$$rp_{if} > 0$$

```
if desiredReward.Amount.IsPositive() {
  // user-controlled minimum ratio of reward to repayment, expressed in base:base
assets (not uTokens)
  rewardTokenEquivalent, err := k.ExchangeUToken(ctx, reward)
  if err != nil {
     return sdk.ZeroInt(), sdk.ZeroInt(), err
  }
```

let $rw_1 = rewardTokenEquivalent.Amount$,

if $desiredReward.Amount > 0 \land err = nil$, we have

$$rw_1 = \lfloor ([urw_1 > urw_{if}] * urw_{if} + [urw_1 \le urw_{if}] * urw_1) * uex \rfloor$$

```
  minimumRewardRatio :=
sdk.NewDecFromInt(desiredReward.Amount).QuoInt(desiredRepayment.Amount)
```

$$minimumRewardRatio = \frac{desiredReward.Amount}{desiredRepayment.Amount}$$

```
  actualRewardRatio :=
sdk.NewDecFromInt(rewardTokenEquivalent.Amount).QuoInt(repayment.Amount)
```

$$actualRewardRatio = \frac{rw_1}{[urw_1 > urw_{if}]*rp_{if} + [urw_1 \le urw_{if}]*rp_0}$$

to simplify the *actualRewardRatio* equation, we examine it under two cases:

Case 1: $urw_1 \le urw_{if}$, we have

$$actualRewardRatio = \frac{urw_1 * uex}{rp_0} = \frac{urw_0 * (1+i) * uex}{rp_0}$$
$$= rp_0 * \frac{TokenPrice[rp.Denom]}{TokenPrice[rw.Denom]} * \frac{1}{uex} * \frac{(1+i) * uex}{rp_0}$$ [7]
$$= \frac{TokenPrice[rp.Denom]}{TokenPrice[rw.Denom]} * (1 + i)$$

Case 2: $urw_1 > urw_{if}$, we have

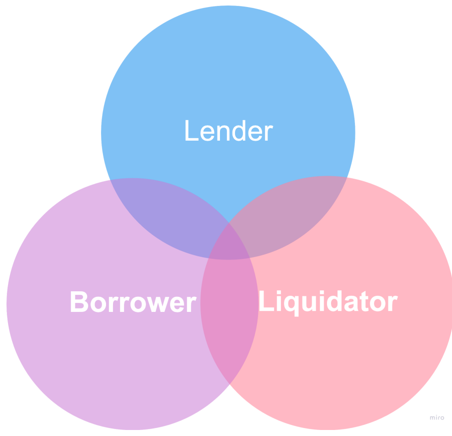
$$actualRewardRatio = \frac{urw_{if} * uex}{rp_{if}} = \frac{urw_{if} * uex}{rp_0 * \frac{urw_{if}}{urw_1}} = \frac{urw_1 * uex}{rp_0}$$
$$= \frac{TokenPrice[rp.Denom]}{TokenPrice[rw.Denom]} * (1 + i)$$

Unify Case 1 and Case 2, we have $actualRewardRatio$ as a constant

$$\frac{TokenPrice[rp.Denom]}{TokenPrice[rw.Denom]} * (1 + i)$$

Leverage Module | Operations

In the leverage module, there are two groups of interactions with it, i.e., the governance account and the user account. So far the governance account is able to update the token registry, a.k.a., the $RegisteredToken$ state variable. The users, *depicted in the graph*[8] *below*, could be the borrower who is allowed to enable a collateral, borrow or repay assets; the lender could lend or withdraw assets while the liquidators will liquidate the eligible borrows and request for a reward. The users could have multiple identities, for example, a lender can be a borrower or a liquidator or both, shown in the venn diagram on the left.

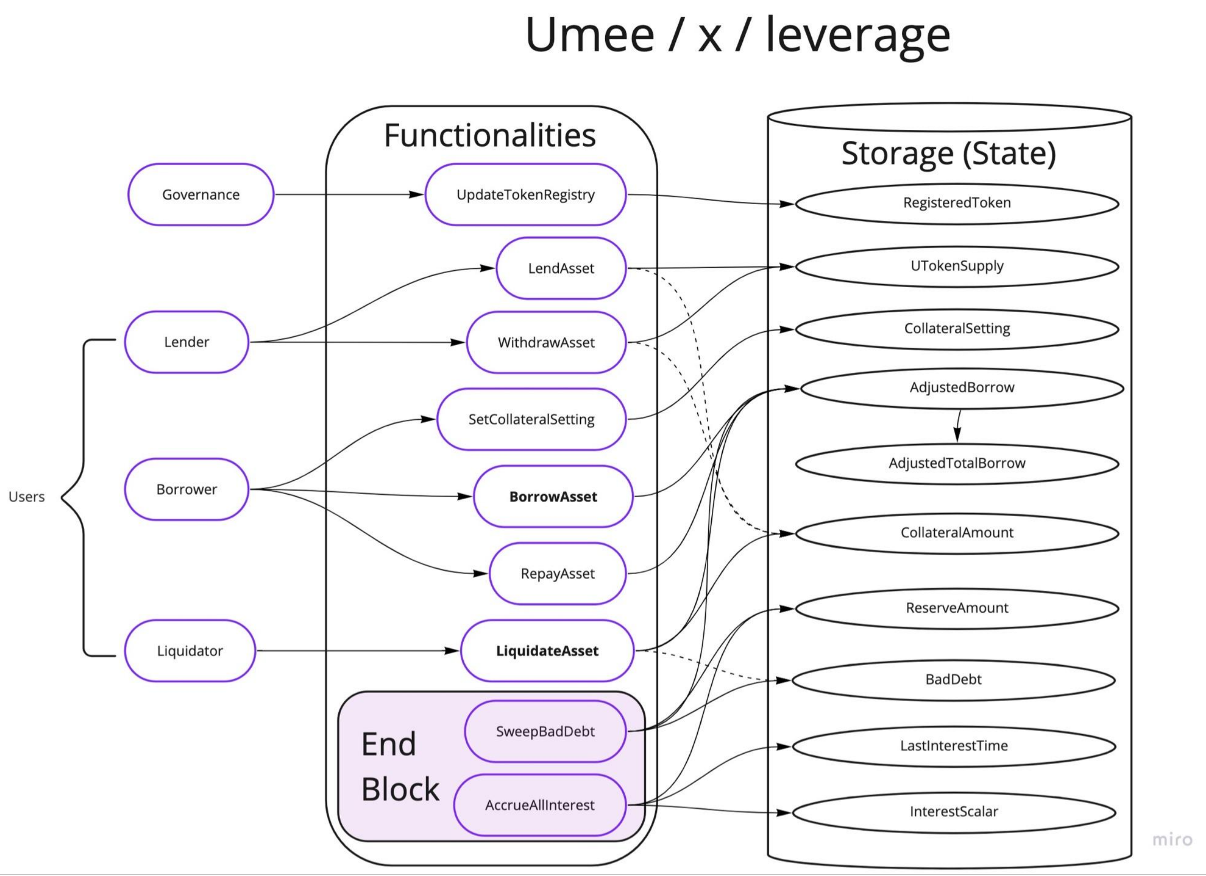


Lending Asset

The lenders send their base tokens to the leverage module in exchange for corresponding uTokens, which can be stored in the user's bank account or collateralized to facilitate future borrows. In the case that if the lender has enabled the token as a collateral, the newly exchanged

[7] Here, we safely remove the floor() function applied on the original expression, assuming the effect is negligible.
[8] In the graph, the arrows representing the variables that are updated by this operation where the solid lines are must happen in a successful event while the dashed lines represent a possible update meeting certain conditions.

uTokens will go to the user's collateral instantly. This event will mint uTokens and increase the



uToken supply.

Withdraw Asset

The lenders are allowed to withdraw base tokens loaned to the leverage module, by exchange uTokens held back to base tokens. The earned lending interest is reflected on the exchange rate that is never decreasing. In the withdrawal event, the user could send the request either in a base token or a uToken, and will only receive base tokens.

Furthermore, it is able to withdraw a combination of uToken balance and collateralized uTokens, however, this should never bring the collatersied value lower than the borrow limit.

Collateral Settings

Borrowers are allowed to enable or disable collaterals through the function *setCollateralSettings*(). In the event that the borrower sets a new collateral, all the available amount of the requested uTokens goes to the leverage module. However, when disabling a collateral, it will only succeed when the borrowed value does not exceed the borrow limit if the uTokens are removed from the collaterals.

Borrow Asset

Borrowers are allowed to borrow new assets, fulfilling the condition that the total borrowed value including the intended borrow should not exceed the borrow limit. The logics are implemented in $x/leverage/borrows.go$.
Assume a borrower b intends to borrow an asset of base token d of amount amt, we have

`Borrow Limit (in USD)`

```go
// CalculateBorrowLimit uses the price oracle to determine the borrow limit (in USD)
provided by
// collateral sdk.Coins, using each token's uToken exchange rate and collateral weight.
// An error is returned if any input coins are not uTokens or if value calculation fails.
func (k Keeper) CalculateBorrowLimit(ctx sdk.Context, collateral sdk.Coins) (sdk.Dec, error)
{
   limit := sdk.ZeroDec()
```

$$limit = 0.0$$

```go
  for _, coin := range collateral {
      // convert uToken collateral to base assets
      baseAsset, err := k.ExchangeUToken(ctx, coin)
      if err != nil {
          return sdk.ZeroDec(), err
      }
```

For $coin \in collateral[b]$, $baseAsset = \lfloor coin.Amount * uExchangeRate[coin.Denom] \rfloor$

```go
      // get USD value of base assets
      value, err := k.TokenValue(ctx, baseAsset)
      if err != nil {
          return sdk.ZeroDec(), err
      }
```

Let $denom = TokenFromUToken(coin.Denom)$
$value = baseAsset * TokenPrice[denom]$[9]

```go
      weight, err := k.GetCollateralWeight(ctx, baseAsset.Denom)
      if err != nil {
          return sdk.ZeroDec(), err
      }
```

[9] Assume TokenPrice[denom] has handled the conversion between base denom and symbol denom.

$$weight = RegisteredToken[denom].CollateralWeight$$

```
        // add each collateral coin's weighted value to borrow limit
        limit = limit.Add(value.Mul(weight))
    }
```

$$limit \mathrel{+}= value * weight$$

```
    return limit, nil
}
```

Thus,

$$Limit = \sum_{coin \in collateral[b]} \lfloor uExchangeRate[coin.Denom] * coin.Amount \rfloor * TokenPrice(denom) * RegisteredToken[denom].CollateralWeight$$

where

$$denom = TokenFromUToken(coin.Denom)$$

`Total Borrowed Value (including new borrow)`

$$TotalBorrowedValue = \lceil amt * TokenPrice[d] \rceil + \sum_{coin \in AdjustedBorrow[b]} \lceil coin.Amount * InterestScalar[coin.Denom] * TokenPrice[coin.Denom] \rceil$$

Repay Asset

Borrowers are able to repay their borrowings through this function. Though borrowers will send a tentative repayment amount, only the owed value is necessary to be paid. Thus, the actually paid amount is no greater than the intended repayment amount.

Liquidate Asset

Liquidators are able to liquidate eligible borrower's collateral, when the borrower's total borrowed value is exceeding the liquidation limit. The liquidator will send the borrower's address *b*, desired repayment *rp* and desired reward *rw* to the leverage module. In the case of a successful liquidation, the liquidator will receive uTokens as rewards.

At the end of a liquidation event, if the borrower's collateral is drained, all the borrowings of the borrower will be labeled as bad debt.

`Liquidation limit:`

$$LiquidationLimit = \sum_{coin \in collateral[b]} coin.Amount * TokenPrice[coin.Denom] * RegisteredToken[coin.Denom].LiquidationThreshold$$

`Liquidation repayment amount:`
The final liquidation repayment amount is bounded by five factors, i.e., the desired amount requested by the liquidator, the liquidator's bank balance of the denom, the actual borrowed amount of the eligible borrower, the maximum amount repayable in a single liquidation event and the collateralized uToken that is available as a reward,

$$\begin{aligned} repayment = min\,(&rp.Amount, \\ &spendableCoins[liquidator][rp.Denom], \\ &borrowed[borrower][rp.Denom], \\ &maxRepaymentAmount, \\ &availableCollateralEquivalent) \end{aligned}$$

Where

$$maxRepaymentAmount = \frac{TotalBorrowedValue*closeFactor}{TokenPrice[denom]}$$

$availableCollateralEquivalent$ is a constant, derived as follows (with code lines 411, 461 - 483 of x/leverage/keeper/keeper.go):

```
    // get borrower uToken balances, for all uToken denoms enabled as collateral
    collateral := k.GetBorrowerCollateral(ctx, borrowerAddr)
```

```
// Given repay denom and amount, use oracle to find equivalent amount of
// rewardDenom's base asset.
baseReward, err := k.EquivalentTokenValue(ctx, repayment, baseRewardDenom)
if err != nil {
  return sdk.ZeroInt(), sdk.ZeroInt(), err
}
```

we know $baseRewardDenom = desiredReward.Denom$

assume $rp_0 = repayment.Amount$, $rw_0 = baseReward.Amount$ both of type $sdk.Coin$

if $err = nil$, we have

$$rw_0 = \lfloor rp_0 * \frac{TokenPrice[rp.Denom]}{TokenPrice[rw.Denom]} \rfloor$$

```
// convert reward tokens back to uTokens
reward, err := k.ExchangeToken(ctx, baseReward)
if err != nil {
  return sdk.ZeroInt(), sdk.ZeroInt(), err
}
```

$$\text{let } urw_0 = reward.Amount,\ uex = \frac{tokenSupply\,(rw.denom)}{uTokenSupply[u/rw.denom]},$$

$$\text{if } err = nil, \text{ we have}$$

$$urw_0 = \left\lfloor \frac{rw_0}{uExchangeRate[rw.Denom]} \right\rfloor$$

```
// apply liquidation incentive
reward.Amount =
reward.Amount.ToDec().Mul(sdk.OneDec().Add(liquidationIncentive)).TruncateInt()
```

$$\text{let } i = RegisteredToken[rw.Denom].LiquidationIncentive$$

$$urw_1 = urw_0 * (1 + i)$$

```
// reward amount cannot exceed available collateral
if reward.Amount.GT(collateral.AmountOf(reward.Denom)) {
  // reduce repayment.Amount to the maximum value permitted by the available
collateral reward
  repayment.Amount =
repayment.Amount.Mul(collateral.AmountOf(reward.Denom)).Quo(reward.Amount)
  // use all collateral of reward denom
  reward.Amount = collateral.AmountOf(reward.Denom)
}
```

$$\text{if } urw_1 > collateral[b][u/rw.Denom], \text{ we have}$$

$$urw_{if} = collateral[b][u/rw.Denom]$$

$$rp_{if} = rp_0 * \frac{urw_{if}}{urw_1} = rp_0 * \frac{collateral[b][u/rw.Denom]}{urw_0 * (1+i)}$$

$$= rp_0 * \frac{collateral[b][u/rw.Denom]}{(1+i)} * \frac{uex}{rw_0}$$

$$= rp_0 * \frac{collateral[b][u/rw.Denom]}{(1+i)} * \frac{uex}{rp_0 * \frac{TokenPrice[rp.Denom]}{TokenPrice[rw.Denom]}}$$

$$= \frac{collateral[b][u/rw.Denom]}{(1+i)} * uExchangeRate[rw.Denom] * \frac{TokenPrice[rw.Denom]}{TokenPrice[rp.Denom]}$$

Thus, the $availableCollateralEquivalent =$

$$\left\lfloor \frac{collateral[b][u/rw.Denom]}{(1+i)} * uExchangeRate[rw.Denom] * \frac{TokenPrice[rw.Denom]}{TokenPrice[rp.Denom]} \right\rfloor$$

Sweep Bad Debt

At the end of each block, this function iterates the $BadDebt$ variable. If a borrower's collateral remains zero, the borrowed asset labeled with bad debt label will have a chance to be repaid using $ReservedAmount$.

Accrue Interest

At the end of each block, interest will be accrued after sweeping bad debt. The newly accumulated interest will be divided into three parts, one part sent to the Oracle module to fund rewarding of the validator, one part saved to the reserved asset to cover bad debt and the rest will be used to pay out lender interest. At the end of this function, the interest scalar is updated which will be used in the next block.

At k^{th} block (k > 1), for a particular denom d, we have its

$$increase[d]_i = r[d]_i * \frac{blocktime_i - blocktime_{i-1}}{secondPerYear}$$

`Total interest accrued`

$$interest[d]_k = TotalBorrowed[d]_k * increase[d]_k$$
$$= adjustedTotalBorrowed[d]_k * interestScalar[d]_{k-1} * increase[d]_k$$

`Newly added reserves`

$$newReserves[d]_k = interest[d]_k * RegisteredTokens[d].reserveFactor$$

`New rewards to fund oracle module`

$$fundIntended[d]_k = interest[d]_k * oracleRewardFactor$$
$$availableBalance[d]_k = moduleBalance[d]_k - newReserves[d]_k$$
$$reward[d]_k = max(min(fundIntended[d]_k, availableBalance[d]_k), 0)$$

Leverage Module | Invariants

The protocol's finance logic is expected to satisfy the following invariants. However, no formal proof is conducted to ensure they are satisfied by the implementation during this audit.

I01: Borrow rate is always between two non-negative values

In fact, it is a nice property if the actual borrow rate has the trend depicted in Findings B02.

I02: $Collateral[borrower][denom] > 0 \rightarrow CollateralSettings[borrower][denom]$

If there are active collaterals, i.e., $Collateral[borrower][denom] > 0$, then the collateral setting for that denom should be true, i.e., $CollateralSettings[borrower][denom]$.

I03: $actualRewardRation \neq 0 \rightarrow actualRewardRation \geq minimumRawardRation$

In a successful liquidation event, where the liquidator does receive a reward, than the actual reward ratio should be no less than the expected ratio, i.e., $minimumRewardRation$.

I04: $totalBorrowInterest \ = \ totalLendingInterest \ + \ totalRewardedtoOracle + \ totalReserved$

It is expected that the leverage module earns enough interest to pay for the lender's lending interest, rewards to validators in the oracle module and reserves to cover bad debt, such that the lending pool won't lose money.

This has brought forward another accompanying property:

$$totalBadDebtPaid \ \leq \ totalReserved.$$

I05: Lending interest rate should be non-negative

I06: uToken exchange rate should never decrease.

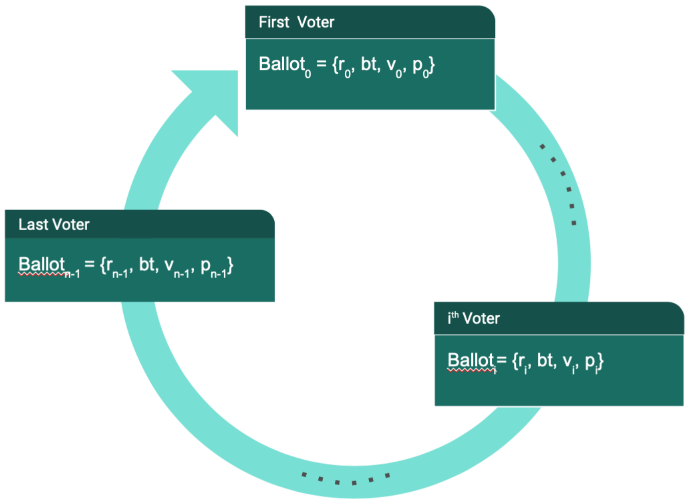
I07: At the end of each block, all active borrowings of a borrower should be labeled bad debt if and only if their collateral is zero.

$$borrowed[borrower]\,!= \ 0 \ \rightarrow$$

$$(\,collateral[borrower] \ == \ 0 \leftrightarrow \wedge_{d\,\in borrowed[borrower]} BadDebt[borrower][d] \ = \ true\,)$$

UMEE Oracle Module | Token[10] Price

In each voting period, the voters will reveal their exchange rate votes for all accepted denominations (in USD, e.g., $(Eth,\ 3000)$ submitted in the previous block (in a secret format). Then the oracle module processes the votes into ballots for each denomination. For each denomination $denom$, a ballot of a voter is a record type consisting of {ExchangeRate: rate, Denom:denom, Voter: voter, Power: power}. In the right graph, Ballot$_i$ denotes the ballot of the ith voter for $denom$.


First Voter
$Ballot_0 = \{r_0, bt, v_0, p_0\}$
Last Voter
$Ballot_{n-1} = \{r_{n-1}, bt, v_{n-1}, p_{n-1}\}$
ith Voter
$Ballot_i = \{r_i, bt, v_i, p_i\}$

The oracle price[11] for $denom$ is decided

[10] "Token" and "denomination" are used interchangeably in this document.

[11] In the oracle module, "exchange rate" and "price" are used interchangeably, both referring to the token's market value in USD.

as the weighted median, i.e., the exchange rate of the i^th voter where the voting power reaches over half of the total voting power of the valid voters (valid voter refers to those whose votes are valid and included as a ballot) in ballots sorted by exchange rates in the ascending order. This mechanism can be abstracted using the following equation:

Let

$$pivot(i) = \sum_{j=0}^{i} ballot_j.Power \geq 1/2 \times \sum_{j=0}^{n} ballot_j.Power$$
$$where\ n = ballot.len() - 1\ and\ i \in [0, n]$$

the price of this denomination, $denom$ is decided as $price(denom) = v_i$ when

$$pivot(i) \wedge (\forall k.\ pivot(k) \wedge k \geq i)$$

Findings

A01: Getters return 0 when they should fail

[Severity: Medium | Difficulty: High | Category: Input Validation]

In some instances a failure to unmarshal during reading from storage leads to returning a 0-value instead of panicking or returning an error.

Examples include the `getAdjustedBorrow` and `getAdjustedTotalBorrowed` functions in `store.go`:

```go
func (k Keeper) getAdjustedBorrow(ctx sdk.Context, addr sdk.AccAddress,
denom string) sdk.Dec {
  key := types.CreateAdjustedBorrowKey(addr, denom)
  adjustedAmount := sdk.ZeroDec()

  if bz := ctx.KVStore(k.storeKey).Get(key); bz != nil {
     if err := adjustedAmount.Unmarshal(bz); err == nil {
        return adjustedAmount
     }
  }

  return sdk.ZeroDec()
}
```

In some instances where the direct storage access and unmarshalling function actually returns an error instead of panicking, the issue of returning zero values occurs higher up the stack. For example, in `DeriveBorrowAPY` in `invariants.go`, if the storage access from `GetRegisteredToken` fails due to an unmarshalling error, `DeriveBorrowAPY` will return 0, which will affect other functions.

Scenario

In the case of `getAdjustedBorrow` this could result in unlimited borrowing. If an issue arises where the stored value can't be correctly unmarshalled, a user will always be reported as having borrowed to a value of 0, meaning that they may keep borrowing.

Recommendation

Getters which are not used as part of the `EndBlocker` function should always panic on unmarshal failure. Getters which are used as part of the `EndBlocker` function should return an error, and every user of that function should propagate that error, ensuring transaction failure, while the EndBlocker should react according to safety principles, for example by halting new borrows and lends until the issue is resolved.

Status

An issue (#861) has been opened on the repo.

A02: May not be possible for users to withdraw as much as is available

[Severity: Low | Difficulty: Medium | Category: Functional Correctness]

If a user tries to withdraw more than the available amounts in the pool, the transaction will fail. However, due to timing and front-running issues, it is not possible to know the available amounts at the time of withdrawal. The user must therefore (perhaps manually) carefully issue transactions in case the pool is close to depleted. Alternatively, only sophisticated users are able to deplete the pool, and only through repeated transactions.

Scenario

Imagine a serious market event with heavy activity and liquidity crunch across many networks. As many bots simultaneously withdraw whatever they can, they see many failed transactions due to available amounts going down, and have to retry, or they intelligently scale their withdraws with exponential back-off. At such a time the robustness of the protocol is more important than under normal circumstances, but this design also causes transaction load to be manifold as high as normal.

Recommendation

Offer an alternative to the user to withdraw as much as is available, in case their desired amount exceeds the amount available to withdraw.

Status

There is no plan to implement this recommendation.

Instead, situations of pulling out much liquidity from the protocol will be mitigated through the following incentives suggested by the Umee team:

- Liquidity mining will require locking up assets for a number of days. Being discussed in PR #858.
- Setting a max borrow utilization per token. This would guarantee that there are some liquidity reserves available for liquidators. Being discussed in PR #926.

A03: Interest calculations are skewed

[Severity: Low | Difficulty: Low | Category: Functional Correctness]

The correct calculation of interest is done with exponents, $(1 + r)^x$, where r is interest rate and x is time elapsed (measured in years). In the `EndBlock` function the chosen method for calculating borrow interest is $(1 + r \cdot x)$, e.g. a binomial approximation, which is suitable for small x. However the error is positive for positive r and x, so the interest calculation skews upward. This means that interest calculated on borrows will consistently be somewhat higher than the target (for that utilization rate) as set by governance.

```
// iterate over all accepted token denominations
for _, token := range tokens {
      // interest is accrued by multiplying each denom's Interest Scalar by
the
      // quantity (borrowAPY * yearsElapsed) + 1
      scalar := k.getInterestScalar(ctx, token.BaseDenom)
      increase := k.DeriveBorrowAPY(ctx, token.BaseDenom).Mul(yearsElapsed)
      if err := k.setInterestScalar(ctx, token.BaseDenom,
scalar.Mul(increase.Add(sdk.OneDec()))); err != nil {
            return err
      }

      [...]
}
```

Another way to put it that the rates set by the protocol refer to annual percentage rates (APR) and not annual percentage yield (APY). These can differ substantially. Since borrow rate compounds automatically, APY is more apt to describe the actual interest to be expected. However, due to its simplicity, APR gives a simpler implementation.

Scenario

A Python simulation for a few different block times (in seconds, with a uniform random variation) seeing what the actual borrow rate over a year would be gives the following results. The code is available in Appendix A.

```
block time   seconds variation   interest rate       actual interest      difference
        20                   5            0.1%          0.100050045%    0.000050045%
         8                   2            0.1%          0.100050019%    0.000050019%
```

```
   2                   1         0.1%       0.100050016%    0.000050016%
  20                   5         3.0%       3.045455222%    0.045455222%
   8                   2         3.0%       3.045454123%    0.045454123%
   2                   1         3.0%       3.045453453%    0.045453453%
  20                   5        20.0%      22.140283875%    2.140283875%
   8                   2        20.0%      22.140276746%    2.140276746%
   2                   1        20.0%      22.140275676%    2.140275676%
  20                   5       120.0%     232.011673383%  112.011673383%
   8                   2       120.0%     232.011721539%  112.011721539%
   2                   1       120.0%     232.011684740%  112.011684740%
```

As can be seen by the result, block time variations have little impact on deviation, and higher interest rates give larger deviations, both in absolute and relative terms.

Recommendation

No mitigation necessary from a security standpoint. Make sure documentation clearly reflects the difference, and that it is taken into account for governance decisions.

Status

An issue (#942) has been opened on the repo.

A04: No way to safely remove support for a token

[Severity: Medium | Difficulty: Medium | Category: Security]

Governance can remove support for a token through governance proposals. However, if they do,

- Debtors and creditors can no longer repay and withdraw that token
- Positions with that token can no longer be liquidated.

Basically, the only safe way to pull support for a token is to first have all positions with it be closed. But in the meantime, there is no way to enforce that more does not get added to the balance sheet. Thus safely removing a token can only happen through governance if it first happens organically.

Governance needs to remove support for tokens under some circumstances, because of security issues in the token, black swan price events, and small-cap danger if a market loses market cap.

Scenario

A token supported by Umee undergoes a governance change and is no longer considered trustworthy in the long term. Umee wants to withdraw support for the token and encourage users to withdraw, without having to liquidate all open borrows using it as collateral. There is no way to do so, and deposits keep coming in and the tokens keep getting borrowed.

Recommendation

In the current implementation, governance can set the collateral weight of the token gradually lower until all loans have been repaid, or immediately to 0 (liquidating all current borrowers), which disincentivizes further lending. It does not, however, prevent borrowing that token.

We recommend giving governance the option to suspend lending and borrowing of a token, while still supporting repaying, withdrawing and liquidating the token. This will mean that the system will monotonically deleverage that token, taking it off its balance sheet.

	Contracts balance sheet	Expands balance sheet
Reduces protocol balance	Withdraw	Borrow
Increases protocol balance	Repay, Liquidate	Lend

Status

Resolved in PR #913 based on recommendation from Runtime Verification. This PR is subject to a future audit to be determined by the Umee team.

A05: Lack of token validation when `SetRegisteredToken()`

[Severity: Medium | Difficulty: Medium | Category: Input Validation]

On line 52 in `keeper/token.go`, in function `SetRegisteredToken(ctx, token)`, only rudimentary validation is performed on the token. Furthermore, there is no validation in `InitGenesis()` and `handleUpdateRegistryProposalHandler()` where `setRegisteredToken()` is called.

```go
func (k Keeper) SetRegisteredToken(ctx sdk.Context, token types.Token) {
	if token.BaseDenom == "" {
		panic("empty base denom")
	}

	store := ctx.KVStore(k.storeKey)
	tokenKey := types.CreateRegisteredTokenKey(token.BaseDenom)

	bz, err := k.cdc.Marshal(&token)
	if err != nil {
		panic(fmt.Sprintf("failed to encode token: %s", err))
	}

	k.hooks.AfterTokenRegistered(ctx, token)
	store.Set(tokenKey, bz)
}
```

Scenario

Some of the token parameters may not meet expectation, for example, `BaseBorrowRate` can be non-negative and `KinkUtilizationRate` is positive and less than 1.0.

Recommendation

Call the `Validate()` function in `types/token.go` before writing the token into the registry.

```go
// In types/token.go:
// Validate performs validation on an Token type returning an error if the
Token
// is invalid.
func (t Token) Validate() error {
	if err := sdk.ValidateDenom(t.BaseDenom); err != nil {
		return err
```

```
        }

        [... many more checks ...]

        if t.LiquidationIncentive.IsNegative() {
                return fmt.Errorf("invalid liquidation incentive: %s",
t.LiquidationIncentive)
        }

        return nil
}
```

Status

An issue (#869) has been opened on the repo.

Resolved in PR #913 based on recommendation from Runtime Verification. This PR is subject to a future audit to be determined by the Umee team.

A06: Possibly incorrect calculation logic of spendable balance

[Severity: High | Difficulty: High | Category: Functional Correctness]

When the leverage module queries for its current balance, it uses `GetBalance()` of the bank module. However, this function will return all tokens the requested address has claimed to, even if they are locked.

On line 144 in `leverage/keeper/keeper.go`, in the function `WithdrawAsset()`, `availableAmount` in module balance refers to the balance that is available to be transferred to the lender. The implemented logic is the module balance of a denom minus its reserved amount, however, the correct logic should be the unlocked module balance (check function `subUnlockedCoins()` in …`sdk/x/bank/keeper/send.go`) of this denom minus its reserved amount. If the module has locked coins, these should not count towards total balance.

There is a similar issue in the `BorrowAsset()` function in `leverage/keeper/keeper.go`, (where `GetBalance()` on the bank module gets called directly, see B07: Reuse helper functions) and in `RepayBadDebt()` in `leverage/keeper/reserves.go`.

Locked balances are a feature of vesting accounts, so the expectation is that the leverage module will not have any locked balance, since it is not a vesting account. However, we have not been able to conclusively determine whether or not it is possible for the leverage module to have locked coins on its balance, or if it could possibly have it in the future, either through having staking options or through malicious action.

Scenario

It could result in a failure of withdrawal (or borrow) caused by insufficient spendable coins in the account of the leverage module, though this failure is not detected by leverage module code.

For example, for a particular coin (either a base token or a uToken) in the leverage module balance, assume its balance is b, locked amount l, its reserved amount in leverage is r, when a lender requests to withdraw an amount of w, the leverage module calculates the $available\ amount = b - r$, while the bank module calculates the $spendableAmount = b - l$. It is possible for $b - r > w \wedge b - l < w$.

Recommendation

When checking if there is sufficient amount to transfer from accounts, use the interface `SpendableCoins()` rather than `GetBalance()`. This includes when borrowing, withdrawing and repaying bad debts. The `ModuleBalance()` function should be redesigned, and the `BorrowAsset()` should also make use of it (see B07: Reuse helper functions).

Status

An issue (#896) has been opened on the repo.

Informative findings

B01: Storage access scattered through many files

[Category: Validation, Coding Practice]

The convention in the code base is to validate stored values at the time when they get stored. This is crucial to ensure the integrity protocol parameters, balances, interest rates, etc., and ensuring transactions that would cause invalid storage updates to fail is preferable to dealing with invalid storage once it has been committed. Still, some functions perform extra validation on their arguments, which may originate from a storage read.

Reasoning about the invariants and correctness of storage is crucial to security, both when auditing, making governance decisions and updating the protocol.

However, direct storage reads and writes are logically scattered across many files. If they were gathered in one file, it would be easier to verify that:

- Keys are not reused for multiple values.
- Correct validation is applied to every storage value before being committed.
- Storage is only ever set and retrieved through one function per key.
- All accesses are made in a uniform way.

Recommendation

Don’t mix storage access methods in files with other methods. Isolate storage accesses in `store.go`.

Status

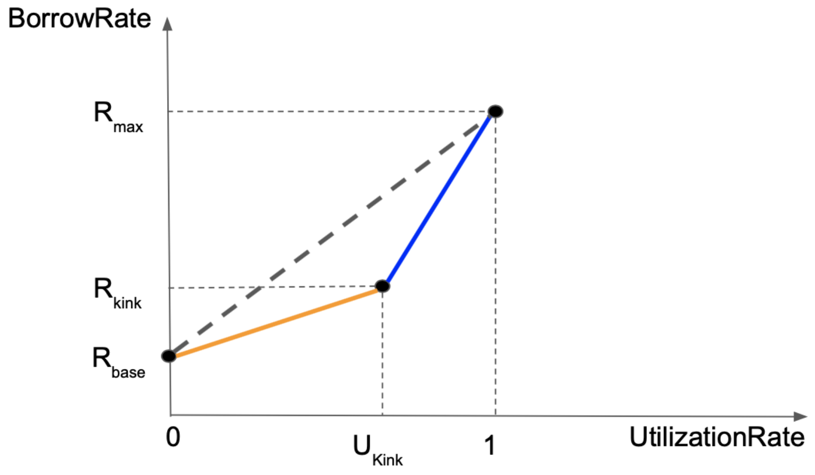
An issue (#862) has been opened on the repo.

B02: Economically unsound parameters are allowed

[Severity: High | Difficulty: Medium | Category: Input Validation, Functional Correctness]

The position of the Umee developers is that *all protocol parameters that are technically feasible should be allowed*. This means that the protocol will not prevent governance from setting parameters that seem clearly unsound, as long as they are technically possible.

It is possible to set borrow rates such that interest decreases with higher utilization. In other money market protocols, for example Aave, the equations governing the interest rates guarantee a monotonic increase in borrow rates. The following curve, in yellow and blue, shows the class of typically acceptable values for the base, kink and max borrow rates, and the kink utilization rate. The curve should be monotonically increasing, with steeper increase after reaching the kink utilization rate. That is, the slope of the dashed line should be positive, and the (U_{kink}, R_{kink}) point (and therefore the yellow and blue curve) should lie below or at the dashed line.



It falls on the shoulders of governance to ensure that any parameters that are set are economically sound. If not, the market dynamics of Umee may be significantly different from those of other decentralized money markets, possibly to detrimental effect.

Recommendation

Document clearly for governance voters that it falls on them to collect and validate reasonable invariants of all protocol parameters they set.

Reasonable invariants:

Borrow rates always increase

$$R_{base} < R_{kink} < R_{max}$$

After reaching kink utilization, borrow rates increase faster

$$\frac{R_{kink} - R_{base}}{U_{kink}} < \frac{R_{max} - R_{kink}}{1 - U_{kink}}$$

or equivalently

$$(R_{kink} - R_{base}) \times (1 - U_{kink}) < (R_{max} - R_{kink}) \times U_{kink}$$

or equivalently

$$R_{kink} - R_{base} < (R_{max} - R_{base}) \times U_{kink}$$

Status

The Umee team intentionally allows governance this freedom of parameter-setting. Risk accepted.

B03: No slippage protection in lend and withdraw functions

[Severity: Medium | Difficulty: High | Category: Functional Correctness]

An invariant of the leverage module is that the exchange rate from an underlying token to its corresponding uToken always increases. However, we have not proved this invariant, so defensive measures against it are still advisable, especially when it directly affects users.

The `WithdrawAsset` function exchanges uTokens for underlying tokens, and specifies either a desired amount of uTokens to burn or a desired amount of underlying tokens to receive, but it does not allow specifying both. The `LendAsset` function exchanges tokens for uTokens, but has no protection if the exchange rate is suddenly manipulated.

Scenario

During unforeseen events, such as when the exchange rate drops due to a bug, slippage can cause burning far more uTokens than a user expected, or lead to them receiving far less underlying tokens then expected.

Recommendation

Users should be aware of this risk, and take care in case of a suspected protocol failure.

The protocol could also mitigate this issue. Either formally verify that the exchange rate never decreases, or implement additional lending and withdrawal functions which allows specifying an acceptable slippage, either by specifying a max amount of tokens burned/minimum amount of tokens received, or a specifying a minimum/maximum exchange rate.

Status

Feature not planned.

B04: `InterestEpoch` documented but not used

[Severity: Low | Difficulty: N/A | Category: Documentation]

The documentation in the `x/oracles` directory and two code comments in the `x/leverage/keeper` directory specify that `InterestEpoch` is the cadence, measured in blocks, at which interest is accrued. However, the implementation accrues interest every block, during `EndBlock`. This means `InterestEpoch` is implicitly set to 1. But it is not a parameter to the protocol.

Furthermore, the use of 1 block as the interest epoch, and using truncated division for calculating interest leads to the actual interest being lower than otherwise expected. However, simulations indicate that this attenuation is extremely small.

Recommendation

Update code comments and documentation.

Status

An issue (#863) has been opened on the repo.

B05: Free loans for a single block

[Severity: Low | Difficulty: Medium | Category: Functional Correctness]

Borrow interest is applied only at the end of the block. Thus, a user can borrow any amount for free for the duration of a single block.

Scenario

The Umee chain adds further DeFi modules. A user submits a bundle of transactions with the expectation that they will all be executed together. The user may be a validator and ensure that the transactions will all happen within one block, or they may ask a validator for the service of including them, or they may simply expect it to happen due to available block real estate and/or transaction fee settings.

The user uses funds in the Umee leverage module to perform an arbitrage on other DeFi modules, then pays the loan back. The user has accrued no interest on their loan.

Recommendation

This issue does not require mitigation from a security perspective. But the protocol may benefit from allowing short-term loans with fixed interest instead – flash-loans – to ensure that such capabilities are available to other users than those with very high collateral amounts. It may also be wise to consider applying a small fixed borrow fee to ensure that every borrow accrues some interest.

Status

Risk accepted.

The idea of a fixed borrow fee is being discussed in an issue on the repo (#943).

B06: Small caps risk

[Severity: High | Difficulty: Variable | Category: Security]

A lending market which calculates collateral value as $latest\ market\ price \cdot tokens\ in\ collateral$ is always vulnerable to small cap token manipulation. Examples of past attacks of this kind are the Cream Finance attack (outlined here by Mudit Gupta) and the Vesper Finance attack (outlined here by their official Twitter account).

The basic problem is as follows: the way collateral value is calculated is the same as how market cap is calculated: $latest\ market\ price \cdot tokens\ tokens\ in\ circulation$. However, it is well known that it would be an error to assume that the total value of a of all the tokens with free floating price is the same as the market cap, in the sense that it would be possible to liquidate them on all open markets, or liquidate them at all, without having significant price impact.

Liquidation mechanisms in money markets rely on the assumption that the protocol or the liquidator will be able to receive more than
$latest\ market\ price \cdot tokens\ in\ collateral \cdot collateral\ weight = allowed\ borrow\ amount$
in exchange for their collateral. This is true when the size of the collateral is small in terms of total market liquidity for the token, but not in general. The pessimistic assumption should be that they should be able to get only as much as the tokens can be immediately sold for in a specific market, which depends on the open buy orders in a traditional market maker or the liquidity positions in an automatic market maker. However, this is hard to encode in a protocol in practice.

The main risk seems to be the ease of manipulating the “real” market value (last price of an observed trade) in tokens with a small market cap. Money market governance must be diligent in setting collateral weights. Another useful mitigation strategy is to limit the amount of borrow allowance that is allowed, protocol-wide, to come from a specific base token, to avoid that the collateral value of those tokens grow large enough to drain significant funds from the protocol due to market manipulation.

The Vesper Finance attack highlights a specific small cap risk to many new money markets: they will likely want to add their own token as collateral, in addition to assets with far higher market caps. The Umee protocol will support the Umee token as collateral, which will likely be small in market cap compared to other assets in the protocol, and thus could be a target for market manipulation.

Scenario

An attacker notices that there is very low liquidity in the available markets used by oracles. They purchase a large number of Umee tokens, causing the market price to double, and deposit them as collateral. The order books now have hollowed out sell sides, and any new sell orders near the current price may not have any natural buyers. This means that the Umee tokens deposited as collateral can only be sold at market for roughly half of the collateral value. If the collateral weight of Umee tokens is 0.75 and the attacker borrows other assets for their full allowance, their borrow amount is now 50% higher than their actual collateral value. If the price of manipulating the token price is less than this discrepancy, they make a profit.

One way to exacerbate the attack is if the orderbooks sell sides are completely depleted. Nothing may prevent the attacker from performing a few sales to themselves (essentially wash trading) at extreme prices once the order books are hollowed out. This way, they may be able to give themselves almost infinite collateral weight once a few oracles price reports have come in and the time weighted price has increased greatly. They can then borrow as much as is available in the protocol, as happened in the Cream Finance and Vesper Finance attacks.

If further financial applications are added on the Umee chain this loop can be made tighter and the attack more efficient.

Recommendation

The time weighted, multi-market approach of the oracles already make these attacks significantly harder.

The difficulty of this attack hinges on the parameters set by governance. It is therefore crucial that tokens accepted as collateral are given sufficiently low collateral weight in accordance with their volatility and manipulability, and that they are only admitted as collateral as long as they have sufficient liquidity on the markets from which prices are fed, and they are available on many markets, and actively traded.

Another mitigation would be to allow governance to cap the amount total borrow allowance that a specific token can provide, a cap which can be freely raised for any organic price increase.

Status

Risk acknowledged, the responsibility for managing the risk is passed on to governance.

B07: Reuse helper functions

[Severity: Low | Difficulty: N/A | Category: Best Practice]

On line 229 in keeper/keeper.go, the get module balance functionality is reimplemented. It is good to reuse the helper function `k.ModuleBalance()` in line 62 of the same file.

```
// ModuleBalance returns the amount of a given token held in the x/leverage
module account
func (k Keeper) ModuleBalance(ctx sdk.Context, denom string) sdk.Int {
      return k.bankKeeper.GetBalance(ctx,
authtypes.NewModuleAddress(types.ModuleName), denom).Amount
}
```

```
      availableAmount := k.bankKeeper.GetBalance(ctx,
authtypes.NewModuleAddress(types.ModuleName), borrow.Denom).Amount
```

Recommendation

Reuse the function `ModuleBalance`.

Status

An issue (#868) has been opened on the repo.

B08: Simplified calculation of maximum allowed repayment

[Severity: Low | Difficulty: N/A | Category: Code Improvement]

The following code in leverage/keeper/keeper.go implements the logic of repayment should not exceed maximum allowed repayment amount.

```
// repayment cannot exceed borrowed value * close factor
maxRepayValue := borrowValue.Mul(closeFactor)
repayValue, err := k.TokenValue(ctx, repayment)
if err != nil {
  return sdk.ZeroInt(), sdk.ZeroInt(), err
}

if repayValue.GT(maxRepayValue) {
  // repayment *= (maxRepayValue / repayValue)
  repayment.Amount =
repayment.Amount.ToDec().Mul(maxRepayValue).Quo(repayValue).TruncateInt()
}
```

in fact, there is no need to calculate *repayvalue*, since `repayment.Amount.ToDec().Mul(maxRepayValue).Quo(repayValue).TruncateInt()` can be interpreted as

$$\frac{repayment.Amount * maxRepayValue}{repayValue},$$

which can be further reduced to

$$\frac{repayment.Amount * maxRepayValue}{repayment.Amount * tokenPrice(repayment.Denom)} = \frac{maxRepayValue}{tokenPrice(repayment.Denom)}.$$

Thus, the repayment amount so far should be the minimum between the current calculated repayment amount and $\lfloor \frac{maxRepayValue}{tokenPrice(repayment.Denom)} \rfloor$.

Recommendation

The simplified code could be

```
denomPrice, err := k.TokenPrice(ctx, repayment.Denom)
if err != nil {
  return sdk.ZeroInt(), sdk.ZeroInt(), err
}

repayment.Amount = sdk.MinInt(repayment.Amount,
maxRepayValue.Quo(denomPrice).TruncateInt())
```

Status

Findings B08 and B09 are listed in the issue #898.

B09: Reward rate is actually a constant in a block

[Severity: Low | Difficulty: N/A | Category: Code Improvement]

The following code makes sure that in a successful liquidation event, the actual reward rate, a.k.a., the amount in the expected reward token versus the amount of the repayment token, is no less than the minimum rate requested by the liquidator.

```
if desiredReward.Amount.IsPositive() {
  // user-controlled minimum ratio of reward to repayment, expressed in
base:base assets (not uTokens)
  rewardTokenEquivalent, err := k.ExchangeUToken(ctx, reward)
  if err != nil {
     return sdk.ZeroInt(), sdk.ZeroInt(), err
  }

  minimumRewardRatio :=
sdk.NewDecFromInt(desiredReward.Amount).QuoInt(desiredRepayment.Amount)
  actualRewardRatio :=
sdk.NewDecFromInt(rewardTokenEquivalent.Amount).QuoInt(repayment.Amount)
  if actualRewardRatio.LT(minimumRewardRatio) {
     return sdk.ZeroInt(), sdk.ZeroInt(), types.ErrLiquidationRewardRatio
  }
}
```

In fact, the actual reward ratio is a constant that can be decided by the desired token prices and the *LiquidationIncentive* parameters of the reward token, as

$$\frac{TokenPrice[desiredRepayment.denom]}{TokenPrice[desiredReward.denom]} * (1.0 + Token[desiredReward.denom].LiquidationIncentive)$$

Please refer to the Description/Rates/RewardRate section of this document for the concrete reasoning.

Recommendation

It is recommended to simplify the LiquidateBorrow() implementation using the above findings. And furthermore, it is recommended to check at the entrance of this function, if the actual reward rate meets the requirements, “no less than minimum ratio” and reject early before any heavy computations take place.

Status

Findings B08 and B09 are listed in the issue #898.

B10: Governance can make user positions insolvent

[Severity: High | Difficulty: Medium | Category: Functional Correctness]

The collateral weight for a token, which controls the borrow allowance of a user using the token as collateral, is set by governance, and can be arbitrarily modified to any value from 0 to 1, inclusive. In particular, it can be set to 0 or otherwise drastically lowered, rendering any user relying on the tokens possibly insolvent, and allowing them to be liquidated.

Recommendation

Users need to be aware of the risk of governance changing collateral weights, and ideally monitor governance proposals in order to manage their risk.

Umee could implement users protections through guaranteeing that the collateral weight is only shifted gradually, with time, towards the goal, giving users reasonable time and allowing monitoring systems to warn about impending liquidation risk. Blocking tokens could be done more directly than setting their collateral weights to 0 (see recommendations A05: No way to safely remove support for a token).

Status

Known risk. No plan to restrict the powers of governance.

B11: `totalInterest` calculated but not used

[Severity: Low | Difficulty: N/A | Category: Best Practice]

On line 104 in `keeper/interest.go`, `totalInterest` is calculated but is not used further apart from in logging. Part of the calculation could be reused later in the function.

```
		// calculate total interest accrued for this denom
		totalInterest = totalInterest.Add(sdk.NewCoin(
			token.BaseDenom,
			prevTotalBorrowed.Mul(increase).TruncateInt(),
		))

		// calculate new reserves accrued for this denom
		newReserves = newReserves.Add(sdk.NewCoin(
			token.BaseDenom,

prevTotalBorrowed.Mul(increase).Mul(token.ReserveFactor).TruncateInt(),
		))

		// calculate oracle rewards accrued for this denom
		oracleRewards = oracleRewards.Add(sdk.NewCoin(
			token.BaseDenom,

prevTotalBorrowed.Mul(increase).Mul(oracleRewardFactor).TruncateInt(),
		))
```

Recommendation

Add a variable `rawInterest := preTotalBorrowed.Mul(increase)` and reuse it in calculating `totalInterest`, `newReserves` and `oracleRewards`.

Status

An issue (#867) has been opened on the repo.

B12: Redundant for loop in `GetExchangeRateBase()`

[Severity: Low | Difficulty: N/A | Category: Best Practice]

On line 110 in `x/oracle/keeper/keeper.go`, this for-loop could be removed.

```
// GetExchangeRateBase gets the consensus exchange rate of an asset
// in the base denom (e.g. ATOM -> uatom)
func (k Keeper) GetExchangeRateBase(ctx sdk.Context, denom string)
(sdk.Dec, error) {
	var symbol string

	// Translate the base denom -> symbol
	params := k.GetParams(ctx)
	for _, listDenom := range params.AcceptList {
		if listDenom.BaseDenom == denom {
			symbol = listDenom.SymbolDenom
			break
		}
	}
	if len(symbol) == 0 {
		return sdk.ZeroDec(), sdkerrors.Wrap(types.ErrUnknownDenom,
denom)
	}

	exchangeRate, err := k.GetExchangeRate(ctx, symbol)
	if err != nil {
		return sdk.ZeroDec(), err
	}

	// NOTE: This for loop can be removed
	for _, acceptedDenom := range params.AcceptList {
		if denom == acceptedDenom.BaseDenom {
			powerReduction :=
ten.Power(uint64(acceptedDenom.Exponent))
			return exchangeRate.Quo(powerReduction), nil
		}
	}

	return sdk.ZeroDec(), sdkerrors.Wrap(types.ErrUnknownDenom, denom)
}
```

Recommendation

Get the exponent in the first for-loop together with the symbol denom. One possible implementation could be: declare a variable `exp` of type `uint64`. Add `exp = listDenon.exponent` right after line 97 (`symbol = …`), before break. Replace line 117 (`return sdk.ZeroDec(), …`) with lines 112 and 113 (`powerReduction := …, return …`).

Status

An issue (#865) has been opened on the repo.

Resolved in PR #866 based on recommendation from Runtime Verification. This PR is subject to a future audit to be determined by the Umee team.

B13: Improve messages when repaying bad debt fails

[Severity: Low | Difficulty: N/A | Category: Best Practice]

On lines 95-111 in `leverage/keeper/reserves.go`, if `newBorrowed` is positive, it could be module balance insufficient or reserves exhausted. Since the repay amount is the minimum of the bad debt amount, the reserved amount or the module balance, if the repay amount cannot cover the bad debt, it could be possible the reserved amount is less than the bad debt or the module balance is less than the bad debt logically.

```
	borrowed := k.GetBorrow(ctx, borrowerAddr, denom)
	reserved := k.GetReserveAmount(ctx, denom)

	amountToRepay := sdk.MinInt(borrowed.Amount, reserved)
	amountToRepay = sdk.MinInt(amountToRepay, k.ModuleBalance(ctx,
denom))

	newBorrowed := borrowed.SubAmount(amountToRepay)

	[...]

	if newBorrowed.IsPositive() {
		k.Logger(ctx).Debug(
			"reserves exhausted",
			"borrower", borrowerAddr.String(),
			"denom", denom,
			"amount", newBorrowed.Amount.String(),
		)
```

Recommendation

Differentiate the logs based on the cases of the insufficient module balance or reserves exhausted.

Status

An issue (#944) has been opened on the repo.

B14: Debt can be bad without the protocol noticing

[Severity: High | Difficulty: Medium | Category: Functional Correctness]

Bad debt is debt that will not get repaid, because it is under-collateralized. It may also not be liquidated, unless some of the debt is first paid off using protocol reserves, making liquidations profitable.

Under normal operating conditions, liquidations prevent under-collateralization. As a second line of defense, protocol reserves are used to pay off the debt once the user has no collateral left. Since liquidators are incentivized to repay debts for undercollateralized loans in exchange for the collateral, it is assumed that collateral will reach 0 in all cases.

However, if positions do not get fully liquidated (i.e. some dust of collateral gets left, not enough for a liquidator to be incentivized to send a liquidation transaction) then the position will remain in the system without being marked as bad debt and cleared. This would go on until someone (a keeper, for example) liquidates them. This can be exacerbated by oracle failures, network congestion, etc. The more types of collateral a position has, the more likely the failure is, because a position is only labeled as bad if it has 0 collateral balance.

If any collateral, even dust amounts, are left, then the protocol will not use reserves to clear the bad debt, and it will keep accumulating interest, making the uToken exchange rate increase, essentially increasing the protocol's debt to the users. What happens when it is clear that the protocol reserves will not be enough to clear the debt is described in B15: uToken market value may not max exchange rate in case of bad debt lingering.

Scenario

A big price drop in a supported token happens (token goes essentially to 0 immediately) and a lot of debt is now bad. No liquidator cares about taking over the collateral. Even if oracles report a token value of 0, the token will not be labeled as bad debt.

Recommendation

Maintain off-chain keepers which perform liquidations. They do not have to aggressively compete with other optimized liquidators, only make sure that positions that are overdue get cleared. The capital requirements should be small due to market incentives to clear any reasonably profitable debt. But in some scenarios the capital requirement might actually be fairly high (in the case of catastrophic price drop), when the delta of current market price to oracle price is high, the delay to get the collateral off the Umee chain (through bridging) is high enough that the expected loss would be even greater, etc. In this situation, capital requirements would be raised. Note that this may be a cost for the keepers, because if a token stops trading and oracle prices get stuck, it can be that liquidations only come at a fairly significant cost of repayment, whereas the collateral is worthless. Keepers, being off-chain code, may also be susceptible to failure when they are most needed, either through DoS attacks or by being crowded out by higher-priced transactions. Keepers should be running in a decentralized fashions and should have monitoring and alerts for the Umee team's on-call.

Another mitigation is raising the threshold for being considered bad debt. Currently it is at 0. You could instead look at the borrow limit to borrowed amount ratio, and if it's below a certain threshold, (say, 1:10) then mark the debt as bad, and have the protocol repay the debt with reserves and take custody of the collateral. This would remove the collateral requirement for keepers, which could be tricky to maintain in a chaotic scenario. It would also let governance adjust this ratio as necessary. Note that liquidators still have the chance to liquidate themselves, first, and the protocol would only liquidate positions that have not been cleared by liquidators.

Status

The Umee team has indicated that they intend to run off-chain keepers to sweep collateral dust. A discussion about the issue has been held in an issue (#513) on the repo. Previously reported by Trail of Bits, TOB-UMEE-22.

B15: uToken market value may not match exchange rate in case of bad debt lingering

[Severity: High | Difficulty: Medium | Category: Functional Correctness]

The design of the exchange rate is to always increase, as the protocol earns more interest. To ensure that, in the case debt goes bad before liquidations of the debt occur, those debts can be cleared by the protocol, some of the protocol earnings get set aside in order to pay off bad debts. However, if the bad debt exceeds available reserves, it's safe to assume the debt in question will never be repaid by the borrower, and may never be liquidated unless the value of the collateral increases in relation to the debt so that the position is once again over-collateralized. Only when the reserves have increased enough to fully pay off the debt will it be finally cleared. That can take time.

In the meantime, the bad debt is still accruing interest, counting towards the exchange rate of the uToken. A reasonable observer should realize that the protocol exchange rate in this case is optimistic, and that the expected return on the uTokens is not quite where it is set by the protocol. Since uTokens are tradeable on the secondary market, this could reasonably lead to the uTokens getting a significantly lower price on secondary markets than on Umee, because users know that it may take a very long time until they will be able to fully withdraw their investment.

If the bad debt is a significant portion of the amount of the token in question on the Umee balance sheet, then the utilization rate may also remain high, leading to very high interest rates that accumulate but will never be repaid.

For a sophisticated user or bot, it will be clear ahead of time if debt has gone bad and will not be covered by reserves, and this gives them time to exit, leaving other users holding the bad debt. Giving preference to the first to cash out may encourage runs.

Recommendation

This scenario is part of the risk of the protocol's design, and something users should take into account to evaluate their investments.

A partial mitigation would be to stop bad debts from accruing interest.

Socializing the bad debt (making sure that all users incur the loss equally once debt has gone bad and will not be repaid with reserves for a long time, or ever) would break the invariant that uToken exchange rates increase monotonically, so it is not advisable to do this, at least in any naive way. However, other solutions for socializing the bad debt rather than making it a run-for-the-door scenario, leaving some users holding the bad debt while others get off fully, should be explored.

Status

Risk accepted. Risk managed by maintaining large reserves and running keepers (liquidation bots).

B16: Oracle price attack risk description

[Severity: High | Difficulty: High | Category: Security]

As indicated in the Summary, a working assumption of this audit is that validators are decentralized and trustworthy enough that the chain can run reliably, and that they supply correct oracle prices.

For completeness, we describe here what it takes to manipulate price oracles.[12]

To manipulate an arbitrary oracle price, the validator needs >50% of the voting power (controlled by Umee tokens) in the validator set. If they have that, they may report any price they like on any two assets.

Scenario

The Umee protocol is successful and plenty of many hundred million dollars worth of tokens are locked in the protocol, available to borrow.

The price of the Umee token is not directly tied to the locked assets.[13] An external event sends the Umee token price down. An attacker is running one of the large validators and takes this opportunity to buy up further validators and market buy up Umee tokens . The attacker spreads their tokens across several validators to avoid suspicion ahead of time. Once all validators are running, the attacker controls more than 50% of the validator voting power.

The attacker buys a small amount of token X of some kind that is accepted as collateral deposit it as collateral in Umee. Ideally this is a collateral type that is not utilized by many other users.

The attacker manipulates the oracle price feed software and deploys it across their validators, to activate one hour in the future.

Once the new software activates the validators controlled by the attacker start reporting a price for tokens X that is extremely high, enough that the attacker's collateral gives them a borrow allowance matching all the available funds in the protocol after a few blocks. The attacker submits transactions borrowing all the available funds in the protocol. They may update their validator software to block the transactions of other users while they wait for the time-weighted price to go up, and once they perform their attack they give their own transactions preferential

[12] Note that this is quite different from manipulating actual market prices, described in B06: Small caps risk. Manipulating the oracle is getting it to report false prices, other than those present in actual markets, while price manipulation means moving markets in a way that would make honest oracles report the manipulated price.
[13] The price of Umee tokens should be influenced by the amount of borrowed assets, since the oracles receive rewards from protocol interest earnings, but not directly by the total value locked in the protocol (lent assets - borrowed assets).

treatment. Since they will control half the blocks being generated during this time they are likely to succeed.

The attacker immediately bridges their borrowed assets to Ethereum or some other larger chain.

Their Umee tokens are still locked in validators, but are likely to lose their value as the attack is uncovered, and may be considered burnt.

This attack is profitable whenever the cost to buy up half the Umee validators is significantly lower than the value of all the tokens available to borrow in Umee.

Recommendation

Users should beware of this possibility, and take extra care if ever the market cap of Umee goes significantly below the total value locked in Umee.

Status

Risk is acknowledged by Umee.

Appendix A: Interest accrual simulation

```
import random


class Dec:
    """Fixed point number implementation"""
    PREC = 18

    def __init__(self, value):
        self.value = int(value)

    def from_num(x):
        return Dec(x * 10**Dec.PREC)

    def add(x, y):
        return Dec(x.value + y.value)

    def sub(x, y):
        return Dec(x.value - y.value)

    # In the interest accrual function, the relevant multiplication rounds,
the
    # relevant quotient truncates.
    def mul(x, y):
        return Dec(round(x.value * y.value / 10**Dec.PREC))

    def quo(x, y):
        return Dec(x.value * 10**Dec.PREC / y.value)

    # For comparison with the multiplication approach.
    def pow(x, y):
        return Dec(
            (x.value / 10**Dec.PREC)**(y.value / 10**Dec.PREC) *
10**Dec.PREC)

    def lt(self, other):
        return self.value < other.value

    def to_num(self):
        return self.value / 10**Dec.PREC

    def __str__(self):
        return str(self.to_num())


def calculate_interest_mul(interest_rate, years_elapsed):
    # (yearly_interest * ratio_of_year + 1)
    return interest_rate.mul(years_elapsed).add(Dec.from_num(1))


def calculate_interest_pow(interest_rate, years_elapsed):
    # (yearly_interest + 1)^ratio_of_year
    return interest_rate.add(Dec.from_num(1)).pow(years_elapsed)


def run_sim(block_time, block_time_variation, interest_rate):
    """Accrue interest every block using a block time as specified, with a
    uniform variation of the block time of +- block_time_variation.  Return
the
    interest rate calculated through binomial approximation and the
interest
    rate calculated using the "correct", power based calculation."""
    block_time = Dec.from_num(block_time)
    interest_rate = Dec.from_num(interest_rate)

    sec_per_year = Dec.from_num(365 * 24 * 60 * 60)

    total_time_elapsed = Dec(0)
    interest_scalar = Dec.from_num(1)
    interest_scalar_real = Dec.from_num(1)
    while total_time_elapsed.lt(sec_per_year):
        variance = Dec.from_num(block_time_variation * random.uniform(-1,
1))
        seconds_elapsed = block_time.add(variance)
        total_time_elapsed = total_time_elapsed.add(seconds_elapsed)

        years_elapsed = seconds_elapsed.quo(sec_per_year)

        interest_scalar = interest_scalar.mul(
            calculate_interest_mul(interest_rate, years_elapsed))
        interest_scalar_real = interest_scalar_real.mul(
            calculate_interest_pow(interest_rate, years_elapsed))

    return (interest_scalar, interest_scalar_real)


def print_sims(params):
    print(
        "block time\tseconds variation\tinterest rate\tactual interest
rate\t  difference\t actual interest (pow)\t  difference"
    )
    while len(params) > 0:
        (block_time, block_time_variation, interest_rate) = params[0]
        params = params[1:]
        (interest_scalar,
         interest_scalar_real) = run_sim(block_time, block_time_variation,
                                         interest_rate)

print("%10d\t%17d\t%12.1f%%\t%19.9f%%\t%10.9f%%\t%21.9f%%\t%10.9f%%" %
              (block_time, block_time_variation, 100 * interest_rate, 100 *
               (interest_scalar.to_num() - 1), 100 * interest_scalar.sub(

Dec.from_num(1)).sub(Dec.from_num(interest_rate)).to_num(),
               100 * (interest_scalar_real.to_num() - 1),
               100 * interest_scalar_real.sub(Dec.from_num(1)).sub(
                   Dec.from_num(interest_rate)).to_num()))


print_sims([
    (20, 5, 0.001),
    (8, 2, 0.001),
    (2, 1, 0.001),
    (20, 5, 0.03),
    (8, 2, 0.03),
    (2, 1, 0.03),
    (20, 5, 0.20),
    (8, 2, 0.20),
    (2, 1, 0.20),
    (20, 5, 1.20),
    (8, 2, 1.20),
    (2, 1, 1.20),
])
```