Security Audit Report for Bitway App Chain

Date: August 29, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction 1

1.1 About Audit Target . . . 1
1.2 Disclaimer . . . 2
1.3 Procedure of Auditing . . . 2
1.3.1 Security Issues . . . 2
1.3.2 Additional Recommendation . . . 3
1.4 Security Model . . . 3

Chapter 2 Findings 5

2.1 Security Issue . . . 6
2.1.1 Lack of distributing liquidation bonuses . . . 6
2.1.2 Loss of funds due to duplicate deposit transactions . . . 8
2.1.3 Prevention of loan repayments . . . 10
2.1.4 Incorrect calculation of the share price . . . 11
2.1.5 Potential DoS due to the lack of status updates . . . 14
2.1.6 Lack of deducting the redundant protocol fee . . . 14
2.1.7 Improper validation of the liquidation bonus factor . . . 15
2.1.8 Lack of checks when updating the configuration `PoolConfig` . . . 16
2.1.9 Potential DoS due to unlimited loan applications . . . 17
2.1.10 Potential runtime panic due to unrestricted staking requests . . . 18
2.1.11 Potential runtime panic due to the improper update of `RewardPerEpoch` . . . 19
2.1.12 Incorrect reward estimation . . . 20
2.1.13 Potential DoS in the DKG completion process . . . 23
2.1.14 Improper design of disabling the farming module . . . 25
2.1.15 Lack of patching the cosmos-sdk package . . . 26
2.2 Recommendation . . . 26
2.2.1 Review the incorrect formula annotation . . . 26
2.2.2 Revise the typos . . . 27
2.2.3 Add duplicate checks for `Maturity` when configuring the pool's tranches . 27
2.2.4 Remove redundant code . . . 28
2.2.5 Refactor the fee fetching logic . . . 29
2.2.6 Perform proper cleanup in the function `Unstake()` . . . 29
2.3 Note . . . 31
2.3.1 The design of the loan's `Authorizations` field . . . 31
2.3.2 The design of liquidation and bad debt management . . . 32
2.3.3 The design of price queries . . . 33
2.3.4 Potential centralization risks . . . 34

Report Manifest

Item	Description
Client	Bitway Labs
Target	Bitway App Chain

Version History

Version	Date	Description
1.0	August 29, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-09-01 02:37:24+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Audit Target

Information	Description
Type	Smart Contract
Language	Go
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of Bitway App Chain of Bitway Labs. Note that this repository is rebranded and migrated to a new repository [2].

The Bitway App Chain of the Bitway Lab is a L1 blockchain, facilitating to unlock the value of underutilized Bitcoins. To support full Bitcoin compatibilities, the Bitway App Chain is designed to enable transactions to be signed with standard Bitcoin wallets for flexibility. The Bitway App Chain natively integrate lending and farming services, powered by Discreet Log Contracts (i.e., DLC). For the Bitcoin-collateralized lending system of the Bitway App Chain, it enables native Bitcoin-backed loans, integrating decentralized oracles (i.e., Oracle++), permissionless liquidity pools, and liquidations. Specifically, The Bitway App Chain implements a liquidity pool-based lending protocol that allows Bitcoin holders to borrow while enabling lenders to earn returns. Loan assets are managed on the Bitway App Chain, removing the need for third-party custodians during the loan period. In addition to lending, the Bitway App Chain also provides a farming service that rewards users for staking their coins. By participating in farming, users can lock their assets and earn incentives, distributed on an epoch basis.

Note this audit focuses on the smart contracts located in the side/x/dlc, side/x/lending, side/x/liquidation, side/x/oracle and side/x/farming directories, excluding the following directories/files:

- *_test.go
- *_simulation.go
- *.pb.go
- *.pb.gw.go

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

[1] https://github.com/sideprotocol/side.git

[2] https://github.com/bitwaylabs/bitway.git

Project	Version	Commit Hash
Bitway App Chain	Version 1	a3eaa2e70b02eee3fad3c0bb8804a276a3ead0a5
	Version 2	ef8fb91bf93ed955f4a61b7614cff4373c65d761
	Version 3	9d515f9df2687558eb88db456b6ece4004c9f833

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)

* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**,

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

Medium, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **fifteen** potential security issues. Besides, we have **six** recommendations and **four** notes.

- High Risk: 2
- Medium Risk: 4
- Low Risk: 9
- Recommendation: 6
- Note: 4

ID	Severity	Description	Category	Status
1	High	Lack of distributing liquidation bonuses	Security Issue	Fixed
2	High	Loss of funds due to duplicate deposit transactions	Security Issue	Fixed
3	Medium	Prevention of loan repayments	Security Issue	Fixed
4	Medium	Incorrect calculation of the share price	Security Issue	Confirmed
5	Medium	Potential DoS due to the lack of status updates	Security Issue	Fixed
6	Medium	Lack of deducting the redundant protocol fee	Security Issue	Fixed
7	Low	Improper validation of the liquidation bonus factor	Security Issue	Fixed
8	Low	Lack of checks when updating the configuration `PoolConfig`	Security Issue	Fixed
9	Low	Potential DoS due to unlimited loan applications	Security Issue	Fixed
10	Low	Potential runtime panic due to unrestricted staking requests	Security Issue	Fixed
11	Low	Potential runtime panic due to the improper update of `RewardPerEpoch`	Security Issue	Fixed
12	Low	Incorrect reward estimation	Security Issue	Fixed
13	Low	Potential DoS in the DKG completion process	Security Issue	Confirmed
14	Low	Improper design of disabling the farming module	Security Issue	Fixed
15	Low	Lack of patching the cosmos-sdk package	Security Issue	Fixed
16	-	Review the incorrect formula annotation	Recommendation	Fixed
17	-	Revise the typos	Recommendation	Fixed
18	-	Add duplicate checks for `Maturity` when configuring the pool's tranches	Recommendation	Fixed
19	-	Remove redundant code	Recommendation	Confirmed

20	-	Refactor the fee fetching logic	Recommendation	Fixed
21	-	Perform proper cleanup in the function `Unstake()`	Recommendation	Confirmed
22	-	The design of the loan's `Authorizations` field	Note	-
23	-	The design of liquidation and bad debt management	Note	-
24	-	The design of price queries	Note	-
25	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of distributing liquidation bonuses

Severity High

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the file `liquidation.go`, the function `HandleLiquidation()` handles users' liquidation requests and records their liquidation bonuses via the `SetLiquidationRecord()` function for further distributions. In the file `abci.go`, the function `handleCompletedLiquidations()` finalizes completed liquidations via building the settlement transactions. Specifically, the settlement transactions send the liquidated collateral to liquidators on the `BTC` network. However, there is a lack of distributing bonuses during the construction of settlement transactions. As a result, liquidators can not receive bonuses via the liquidation.

```
// calculate bonus
bonusAmountInDebt := debtAmount.Amount.Mul(sdkmath.NewInt(int64(k.LiquidationBonusFactor(ctx))))
    .Quo(sdkmath.NewInt(1000))
bonusAmount := types.GetCollateralAmount(bonusAmountInDebt, debtDecimals, collateralDecimals,
    currentPrice, collateralIsBaseAsset)

// check if there is left collateral for bonus
if bonusAmount.GT(remainingCollateralAmount.Amount) {
  bonusAmount = remainingCollateralAmount.Amount
}

// check if the total received collateral amount is dust
if types.IsDustOut(collateralAmount.Add(bonusAmount).Int64(), liquidator) {
  return nil, errorsmod.Wrapf(types.ErrInvalidAmount, "dust collateral amount %s",
      collateralAmount)
}

protocolLiquidationFee := bonusAmount.Mul(sdkmath.NewInt(int64(k.ProtocolLiquidationFeeFactor(
    ctx)))).Quo(sdkmath.NewInt(1000))
```

```go

liquidation.LiquidatedCollateralAmount = liquidation.LiquidatedCollateralAmount.AddAmount(collateralAmount).AddAmount(bonusAmount)
liquidation.LiquidatedDebtAmount = liquidation.LiquidatedDebtAmount.Add(debtAmount)
liquidation.LiquidationBonusAmount = liquidation.LiquidationBonusAmount.AddAmount(bonusAmount)
liquidation.ProtocolLiquidationFee = liquidation.ProtocolLiquidationFee.AddAmount(protocolLiquidationFee)
liquidation.UnliquidatedCollateralAmount = liquidation.UnliquidatedCollateralAmount.SubAmount(collateralAmount).SubAmount(bonusAmount)

record := &types.LiquidationRecord{
  Id:               k.IncrementLiquidationRecordId(ctx),
  LiquidationId:    liquidationId,
  Liquidator:       liquidator,
  DebtAmount:       debtAmount,
  CollateralAmount: sdk.NewCoin(liquidation.CollateralAsset.Denom, collateralAmount),
  BonusAmount:      sdk.NewCoin(liquidation.CollateralAsset.Denom, bonusAmount.Sub(protocolLiquidationFee)),
  Time:             ctx.BlockTime(),
}

k.SetLiquidation(ctx, liquidation)
k.SetLiquidationRecord(ctx, record)

return record, nil
```

Listing 2.1: side/x/liquidation/keeper/liquidation.go

```go
  settlementTx, txHash, sigHashes, changeAmount, err := types.BuildSettlementTransaction(liquidation, k.GetLiquidationRecords(ctx, liquidation.Id), k.ProtocolLiquidationFeeCollector(ctx), feeRate.Value)
  if err != nil {
    k.Logger(ctx).Info("Failed to build settlement transaction", "liquidation id", liquidation.Id, "fee rate", feeRate.Value, "err", err)

    continue
  }
```

Listing 2.2: side/x/liquidation/module/abci.go

```go
func BuildSettlementTransaction(liquidation *Liquidation, records []*LiquidationRecord, protocolFeeCollector string, feeRate int64) (string, *chainhash.Hash, []string, int64, error) {
liquidationCet, err := psbt.NewFromRawBytes(bytes.NewReader([]byte(liquidation.LiquidationCet)), true)
if err != nil {
  return "", nil, nil, 0, err
}

txOut := liquidationCet.UnsignedTx.TxOut[0]

utxo := &btcbridgetypes.UTXO{
  Txid:         liquidationCet.UnsignedTx.TxHash().String(),
```

```
  Vout:        0,
  Amount:      uint64(txOut.Value),
  PubKeyScript: txOut.PkScript,
}

settlementTxPsbt, changeAmount, err := BuildBatchTransferPsbt([]*btcbridgetypes.UTXO{utxo},
    records, protocolFeeCollector, liquidation.ProtocolLiquidationFee.Amount.Int64(), feeRate,
    liquidation.Debtor)
if err != nil {
  return "", nil, nil, 0, err
}
```

Listing 2.3: side/x/liquidation/types/bitcoin.go

```

for _, record := range records {
  address, err := btcutil.DecodeAddress(record.Liquidator, chainCfg)
  if err != nil {
    return nil, 0, err
  }

  pkScript, err := txscript.PayToAddrScript(address)
  if err != nil {
    return nil, 0, err
  }

  txOuts = append(txOuts, wire.NewTxOut(record.CollateralAmount.Amount.Int64(), pkScript))
}
```

Listing 2.4: side/x/liquidation/types/bitcoin.go

Impact Liquidators can not receive bonuses via the liquidation.

Suggestion Revise the logic accordingly.

2.1.2 Loss of funds due to duplicate deposit transactions

Severity High

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the file `msg_server_loan.go`, the function `SubmitCets()` allows borrowers to submit `CET`s with deposit transactions on the `BTC` network. Specifically, the collateral amount is accumulated based on provided deposit transactions and set to the borrower's loan. However, there is a lack of duplicate checks for the provided deposit transactions (i.e., `msg.DepositTxs`). This flaw design allows a malicious borrower to submit duplicate deposit transactions with customized `CET`s, leading to a large collateral amount (i.e., `loan.CollateralAmount`) stored in the borrower's loan. As a result, the malicious borrower could drain pools after their loan is approved.

```
collateralAmount := sdkmath.ZeroInt()

```

```go
for _, depositTx := range msg.DepositTxs {
  p, _ := psbt.NewFromRawBytes(bytes.NewReader([]byte(depositTx)), true)

  depositTxs = append(depositTxs, p)
  depositTxHashes = append(depositTxHashes, p.UnsignedTx.TxHash().String())

  for _, out := range p.UnsignedTx.TxOut {
    if bytes.Equal(out.PkScript, vaultPkScript) {
      collateralAmount = collateralAmount.Add(sdkmath.NewInt(out.Value))
    }
  }
}

if collateralAmount.IsZero() {
  return nil, errorsmod.Wrap(types.ErrInsufficientCollateral, "collateral amount can not be zero")
}

dlcEvent := m.dlcKeeper.GetEvent(ctx, loan.DlcEventId)

// verify cets
if err := types.VerifyCets(depositTxs, vaultPkScript, loan.BorrowerPubKey, loan.BorrowerAuthPubKey, loan.DCM, dlcEvent, msg.LiquidationCet, msg.LiquidationAdaptorSignatures, msg.DefaultLiquidationAdaptorSignatures, msg.RepaymentCet, msg.RepaymentSignatures); err != nil {
  return nil, err
}

// update dlc meta
if err := m.UpdateDLCMeta(ctx, msg.LoanId, depositTxs, msg.LiquidationCet, msg.LiquidationAdaptorSignatures, msg.DefaultLiquidationAdaptorSignatures, msg.RepaymentCet, msg.RepaymentSignatures); err != nil {
  return nil, err
}

// create authorization
authorization := m.CreateAuthorization(ctx, msg.LoanId, depositTxHashes)

for i, depositTx := range msg.DepositTxs {
  if !m.HasDepositLog(ctx, depositTxHashes[i]) {
    depositLog := &types.DepositLog{
      Txid:            depositTxHashes[i],
      VaultAddress:    loan.VaultAddress,
      AuthorizationId: authorization.Id,
      DepositTx:       depositTx,
    }

    m.SetDepositLog(ctx, depositLog)
  }
}

collateralDecimals := int(poolConfig.CollateralAsset.Decimals)
borrowDecimals := int(poolConfig.LendingAsset.Decimals)
```

```
collateralIsBaseAsset := poolConfig.CollateralAsset.IsBasePriceAsset

// calculate liquidation price
liquidationPrice := types.GetLiquidationPrice(collateralAmount, collateralDecimals, loan.
    BorrowAmount.Amount, borrowDecimals, loan.Maturity, loan.BorrowAPR, m.GetBlocksPerYear(ctx),
     poolConfig.LiquidationThreshold, collateralIsBaseAsset)

// update loan
loan.Authorizations = append(loan.Authorizations, *authorization)
loan.CollateralAmount = collateralAmount
loan.LiquidationPrice = liquidationPrice
loan.Status = types.LoanStatus_Authorized
m.SetLoan(ctx, loan)

return &types.MsgSubmitCetsResponse{}, nil
}
```

Listing 2.5: side/x/lending/keeper/msg_server_loan.go

Impact The malicious borrower could drain pools with little collateral with duplicate deposit transactions.

Suggestion Revise the logic accordingly.

2.1.3 Prevention of loan repayments

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 1`

Description The function `handleApproval()` approves a loan and requests the `DCM` to pre-sign an adaptor signature for the repayment `CET`. If the borrower repays the loan, the function `handleRepayments()` requires the `ESN` to sign to reveal the adaptor secret. With the revealed secret, the function can convert the `DCM` signature to a complete Schnorr signature. The repayment `CET` can then be fully signed and executed with signatures from both the borrower and `DCM`, enabling borrowers to retrieve their collateral from the vault. However, this pre-signing mechanism contains a critical timing vulnerability. After loan approval, malicious `DCM` participants can refuse to pre-sign, preventing the generation of a valid `DCM` adaptor signature. As a result, this refusal prevents borrowers from receiving the collateral due to the miss of `DCMAdaptorSignatures` (Line 357 in the function `handleRepayments()`).

```
func (k Keeper) HandleApproval(ctx sdk.Context, loan *types.Loan) error {
pool := k.GetPool(ctx, loan.PoolId)
if pool.AvailableAmount.LT(loan.BorrowAmount.Amount) {
  return types.ErrInsufficientLiquidity
}

amount := sdk.NewCoin(loan.BorrowAmount.Denom, loan.BorrowAmount.Amount.Sub(loan.OriginationFee)
    )
if err := k.bankKeeper.SendCoinsFromModuleToAccount(ctx, types.ModuleName, sdk.
    MustAccAddressFromBech32(loan.Borrower), sdk.NewCoins(amount)); err != nil {
```

```
    return err
}

if loan.OriginationFee.IsPositive() {
    originationFee := sdk.NewCoin(loan.BorrowAmount.Denom, loan.OriginationFee)
    if err := k.bankKeeper.SendCoinsFromModuleToAccount(ctx, types.ModuleName, sdk.
        MustAccAddressFromBech32(k.OriginationFeeCollector(ctx)), sdk.NewCoins(originationFee));
        err != nil {
        return err
    }
}

// initiate signing request for repayment cet adaptor signatures from DCM
if err := k.InitiateRepaymentCetSigningRequest(ctx, loan.VaultAddress); err != nil {
    return err
}
```

Listing 2.6: side/x/lending/keeper/approval.go

```
func handleRepayments(ctx sdk.Context, k keeper.Keeper) {
// get all repaid loans
loans := k.GetLoans(ctx, types.LoanStatus_Repaid)

for _, loan := range loans {
    // trigger dlc event if not triggered yet
    if !k.DLCKeeper().GetEvent(ctx, loan.DlcEventId).HasTriggered {
        k.DLCKeeper().TriggerDLCEvent(ctx, loan.DlcEventId, types.RepaidOutcomeIndex)
        continue
    }

    // check if the repayment cet has been signed
    dlcMeta := k.GetDLCMeta(ctx, loan.VaultAddress)
    if len(dlcMeta.RepaymentCet.SignedTxHex) != 0 {
        continue
    }

    // check if the DCM adaptor signatures have been submitted
    if len(dlcMeta.RepaymentCet.DCMAdaptorSignatures) == 0 {
        continue
    }
```

Listing 2.7: side/x/lending/module/abci.go

Impact Borrowers may be prevented from retrieving collateral due to the miss of the signature `DCMAdaptorSignatures`.

Suggestion Revise the logic to obtain `DCM` adaptor signature submission before loan approval.

2.1.4 Incorrect calculation of the share price

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the file `pool.go`, the functions `GetSTokenAmount()` and `GetUnderlyingAssetAmount()` calculate the amount of shares and assets based on the variable `pool.TotalBorrowed`. During the liquidation process, the interest of the liquidated loan continues to accrue and is accumulated in the variable `pool.TotalBorrowed`. After the liquidation is completed, in the function `HandleLiquidatedDebt()`, the redundant interest will be deducted from the variable `pool.TotalBorrowed` via the function `DeductLiquidationAccruedInterest()`. However, this design could impact the calculations of the functions `GetSTokenAmount()` and `GetUnderlyingAssetAmount()`, which include redundant interest that has yet to be deducted. As a result, users may receive an inaccurate amount of shares and assets due to the redundant interest.

```go

// GetSTokenAmount calculates the sToken amount from the given deposit amount
func (k Keeper) GetSTokenAmount(ctx sdk.Context, pool *types.LendingPool, depositAmount sdkmath.
    Int) sdkmath.Int {
  return depositAmount.Mul(pool.TotalSTokens.Amount).Quo(pool.AvailableAmount.Add(pool.
      TotalBorrowed).Sub(pool.TotalReserve))
}

// GetUnderlyingAssetAmount calculates the underlying asset amount from the given sToken amount
func (k Keeper) GetUnderlyingAssetAmount(ctx sdk.Context, pool *types.LendingPool, sTokenAmount
    sdkmath.Int) sdkmath.Int {
  return sTokenAmount.Mul(pool.AvailableAmount.Add(pool.TotalBorrowed).Sub(pool.TotalReserve)).Quo
      (pool.TotalSTokens.Amount)
}
```

Listing 2.8: side/x/lending/keeper/pool.go

```go
func (k Keeper) HandleLiquidatedDebt(ctx sdk.Context, liquidationId uint64, loanId string,
    moduleAccount string, debtAmount sdk.Coin) error {
  loan := k.GetLoan(ctx, loanId)
  pool := k.GetPool(ctx, loan.PoolId)

  interest := k.GetCurrentInterest(ctx, loan).Amount

  principal := sdk.NewCoin(debtAmount.Denom, sdkmath.ZeroInt())
  if debtAmount.Amount.GT(interest) {
    // split debt to principal and interest
    principal = debtAmount.SubAmount(interest)
  } else {
    // consider debt as interest
    interest = debtAmount.Amount
  }

  protocolFee := types.GetProtocolFee(interest, pool.Config.ReserveFactor)

  referralFee := sdkmath.ZeroInt()
  actualProtocolFee := protocolFee
  if protocolFee.IsPositive() && types.HasReferralFee(loan, pool) {
    referralFee = protocolFee.Mul(sdkmath.NewInt(int64(pool.Config.ReferralFeeFactor))).Quo(types.
        Permille)
```

```go
    actualProtocolFee = protocolFee.Sub(referralFee)
  }

  if err := k.bankKeeper.SendCoinsFromModuleToModule(ctx, moduleAccount, types.ModuleName, sdk.
      NewCoins(debtAmount.SubAmount(protocolFee))); err != nil {
    return err
  }

  if actualProtocolFee.IsPositive() {
    if err := k.bankKeeper.SendCoinsFromModuleToAccount(ctx, moduleAccount, sdk.
        MustAccAddressFromBech32(k.ProtocolFeeCollector(ctx)), sdk.NewCoins(sdk.NewCoin(debtAmount
        .Denom, actualProtocolFee))); err != nil {
      return err
    }
  }

  if referralFee.IsPositive() {
    if err := k.bankKeeper.SendCoinsFromModuleToAccount(ctx, moduleAccount, sdk.
        MustAccAddressFromBech32(loan.Referrer), sdk.NewCoins(sdk.NewCoin(debtAmount.Denom,
        referralFee))); err != nil {
      return err
    }
  }

  k.AfterPoolRepaid(ctx, loan.PoolId, loan.Maturity, principal, interest, protocolFee,
      actualProtocolFee)

  k.DeductLiquidationAccruedInterest(ctx, loan)

  return nil
}

// DeductLiquidationAccruedInterest deducts the interest accrued during the loan liquidation from
    total borrowed
func (k Keeper) DeductLiquidationAccruedInterest(ctx sdk.Context, loan *types.Loan) {
  interest := k.GetLiquidationAccruedInterest(ctx, loan)

  k.DecreaseTotalBorrowed(ctx, loan.PoolId, loan.Maturity, interest)
}

// GetLiquidationAccruedInterest gets the current accrued interest during the loan liquidation
func (k Keeper) GetLiquidationAccruedInterest(ctx sdk.Context, loan *types.Loan) sdkmath.Int {
  currentTotalInterest := types.GetInterest(loan.BorrowAmount.Amount, loan.StartBorrowIndex, k.
      GetCurrentBorrowIndex(ctx, loan))

  return currentTotalInterest.Sub(k.GetCurrentInterest(ctx, loan).Amount)
}
```

Listing 2.9: side/x/lending/keeper/liquidation.go

Impact Incorrect share prices due to the redundant interest.

Suggestion Revise the logic accordingly.

Feedback from the project The project stated that they are aware of this issue and they added a field in the struct liquidation to monitor the interest accrued during liquidations.

2.1.5 Potential DoS due to the lack of status updates

Severity Medium

Status Fixed in Version 3

Introduced by Version 1

Description In the file abci.go, the functions handleLiquidatedLoans() and handleDefaultedLoans() handles all defaulted and liquidated loans at the end of the block. However, there is a lack of status update for all defaulted and liquidated loans. All completed defaulted and liquidated loans, which are completed, will be processed again in the functions handleLiquidatedLoans() and handleDefaultedLoans(). As a result, the lack of status updates may lead to a potential DoS issue by increasing the chain's workload.

```
func handleLiquidatedLoans(ctx sdk.Context, k keeper.Keeper) {
    // get all liquidated loans
    loans := k.GetLoans(ctx, types.LoanStatus_Liquidated)
```

Listing 2.10: side/x/lending/module/abci.go

```
func handleDefaultedLoans(ctx sdk.Context, k keeper.Keeper) {
    // get all defaulted loans
    loans := k.GetLoans(ctx, types.LoanStatus_Defaulted)
```

Listing 2.11: side/x/lending/module/abci.go

Impact The lack of status updates may lead to a potential DoS issue by increasing the chain's workload.

Suggestion Revise the logic accordingly.

2.1.6 Lack of deducting the redundant protocol fee

Severity Medium

Status Fixed in Version 3

Introduced by Version 1

Description In the file liquidation.go, the function HandleLiquidatedDebt() removes the redundant interest (i.e., accrued based on the liquidated loan) from the variable pool.TotalBorrowed via the function DeductLiquidationAccruedInterest(). However, it does not deduct the corresponding protocolFee, which is generated based on the redundant interest, from the variable pool.TotalReserve. The lack of deducting the redundant protocol fee enlarges the variable pool.TotalReserve. As a result, users may receive an inaccurate amount of shares and assets due to the redundant protocol fee.

```
    k.AfterPoolRepaid(ctx, loan.PoolId, loan.Maturity, principal, interest, protocolFee,
        actualProtocolFee)

```

```go
	k.DeductLiquidationAccruedInterest(ctx, loan)

	return nil
}
```

Listing 2.12: side/x/lending/keeper/liquidation.go

```go
// DeductLiquidationAccruedInterest deducts the interest accrued during the loan liquidation from
    total borrowed
func (k Keeper) DeductLiquidationAccruedInterest(ctx sdk.Context, loan *types.Loan) {
	interest := k.GetLiquidationAccruedInterest(ctx, loan)

	k.DecreaseTotalBorrowed(ctx, loan.PoolId, loan.Maturity, interest)
}

// GetLiquidationAccruedInterest gets the current accrued interest during the loan liquidation
func (k Keeper) GetLiquidationAccruedInterest(ctx sdk.Context, loan *types.Loan) sdkmath.Int {
	currentTotalInterest := types.GetInterest(loan.BorrowAmount.Amount, loan.StartBorrowIndex, k.
		GetCurrentBorrowIndex(ctx, loan))

	return currentTotalInterest.Sub(k.GetCurrentInterest(ctx, loan).Amount)
}
```

Listing 2.13: side/x/lending/keeper/liquidation.go

```go
// DecreaseTotalBorrowed decreases total borrowed by the given amount for the specified pool
func (k Keeper) DecreaseTotalBorrowed(ctx sdk.Context, poolId string, maturity int64, amount
    sdkmath.Int) {
	pool := k.GetPool(ctx, poolId)

	pool.TotalBorrowed = pool.TotalBorrowed.Sub(amount)

	for i, tranche := range pool.Tranches {
		if tranche.Maturity == maturity {
			pool.Tranches[i].TotalBorrowed = pool.Tranches[i].TotalBorrowed.Sub(amount)
			break
		}
	}

	k.NormalizePool(ctx, pool)

	k.SetPool(ctx, pool)
}
```

Listing 2.14: side/x/lending/keeper/pool.go

Impact Users may receive an inaccurate amount of shares and assets due to the redundant protocol fee.

Suggestion Revise the logic accordingly.

2.1.7 Improper validation of the liquidation bonus factor

Severity Low

Status Fixed in Version 3

Introduced by Version 1

Description In the file `params.go`, the function `Validate()` requires the liquidation bonus factor (i.e., `p.LiquidationBonusFactor`) to be within the range of (0, 1000). However, this validation for the liquidation bonus factor is improper, potentially generating bad debts. Specifically, if the liquidation bonus factor is set to a large value, the collateral, which is left after the bonus deduction, may not cover the rest of the debt. As a result, the improper liquidation bonus factor may generate bad debts.

```go
func (p Params) Validate() error {
	if p.MinLiquidationFactor == 0 || p.MinLiquidationFactor >= 1000 {
		return errorsmod.Wrap(ErrInvalidParams, "invalid minimum liquidation factor")
	}

	if p.LiquidationBonusFactor == 0 || p.LiquidationBonusFactor >= 1000 {
		return errorsmod.Wrap(ErrInvalidParams, "invalid liquidation bonus factor")
	}
```

Listing 2.15: side/x/liquidation/types/params.go

Impact The improper liquidation bonus factor may generate bad debts.

Suggestion Revise the validation of the liquidation bonus factor.

2.1.8 Lack of checks when updating the configuration `PoolConfig`

Severity Low

Status Fixed in Version 3

Introduced by Version 1

Description In the file `msg_server_pool.go`, the function `UpdatePoolConfig()` allows the gov module to update the pool's configuration. However, this function lacks checking the important parameters (e.g., collateral and lending assets), which can not be modified. If a malicious proposal is passed, the improper update of the pool configuration may lead to critical impacts (e.g., loss of funds due to the change of the collateral and lending assets).

```go
func (m msgServer) UpdatePoolConfig(goCtx context.Context, msg *types.MsgUpdatePoolConfig) (*types
    .MsgUpdatePoolConfigResponse, error) {
	if m.authority != msg.Authority {
		return nil, errorsmod.Wrapf(govtypes.ErrInvalidSigner, "invalid authority; expected %s, got %s
			", m.authority, msg.Authority)
	}

	if err := msg.ValidateBasic(); err != nil {
		return nil, err
	}

	ctx := sdk.UnwrapSDKContext(goCtx)

	if !m.HasPool(ctx, msg.PoolId) {
		return nil, types.ErrPoolDoesNotExist
```

```go
}

pool := m.GetPool(ctx, msg.PoolId)
m.UpdatePoolStatus(ctx, pool, &msg.Config)

pool.Config = msg.Config
m.SetPool(ctx, pool)

return &types.MsgUpdatePoolConfigResponse{}, nil
}
```

Listing 2.16: side/x/lending/keeper/msg_server_pool.go

Impact Critical impacts (e.g., loss of funds) due to the improper update logic for the pool configuration.

Suggestion Add more checks when updating the pool configuration in the `UpdatePoolConfig()` function.

2.1.9 Potential DoS due to unlimited loan applications

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the file `msg_server_loan.go`, the function `Apply()` allows users to apply loans with valid inputs. The function `handlePendingLoans()` of the file `x/lending/module/abci.go` further processes all applied loans (i.e., loans with the status `LoanStatus_Requested`). However, when there is no request fee (i.e., `types.HasRequestFee(pool) == false`) in a pool, users can apply as many loans as they want to bloat the list of pending loans. As a result, unlimited loan applications may lead to a potential DoS issue by increasing the chain's workload.

```go
// Apply implements types.MsgServer.
func (m msgServer) Apply(goCtx context.Context, msg *types.MsgApply) (*types.MsgApplyResponse,
    error) {
 if err := msg.ValidateBasic(); err != nil {
   return nil, err
 }
```

Listing 2.17: side/x/lending/keeper/msg_server_loan.go

```go
loans := k.GetPendingLoans(ctx)

for _, loan := range loans {
  pool := k.GetPool(ctx, loan.PoolId)
  authorizationId := k.GetAuthorizationId(ctx, loan.VaultAddress)

  // check if the maturity time already reached
  if ctx.BlockTime().Unix() >= loan.MaturityTime {
    rejectHandler(loan, authorizationId, types.ErrMaturityTimeReached)
    continue
  }
```

Listing 2.18: side/x/lending/module/abci.go

Impact Unlimited loan applications may lead to a potential DoS issue by increasing the chain's workload.

Suggestion Revise the logic accordingly.

2.1.10 Potential runtime panic due to unrestricted staking requests

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the file `msg_server.go` of the module `farming`, the function `Stake()` allows users to stake assets to gain rewards. However, this function lacks a minimum amount requirement restricting users from creating unreasonable staking requests (i.e., staking amount is only 1). This design allows malicious users to request as many small stakings as they want to create a heavy workload for the chain. As a result, the chain may face a runtime panic risk during executing the function `EndBlocker()` due to the unrestricted staking requests.

```
func (m msgServer) Stake(goCtx context.Context, msg *types.MsgStake) (*types.MsgStakeResponse, error) {
	if err := msg.ValidateBasic(); err != nil {
		return nil, err
	}

	ctx := sdk.UnwrapSDKContext(goCtx)

	if !m.FarmingEnabled(ctx) {
		return nil, types.ErrFarmingNotEnabled
	}

	if !m.IsEligibleAsset(ctx, msg.Amount.Denom) {
		return nil, errorsmod.Wrapf(types.ErrAssetNotEligible, "asset %s not eligible", msg.Amount.Denom)
	}

	if !m.LockDurationExists(ctx, msg.LockDuration) {
		return nil, types.ErrInvalidLockDuration
	}

	if err := m.bankKeeper.SendCoinsFromAccountToModule(ctx, sdk.MustAccAddressFromBech32(msg.Staker), types.ModuleName, sdk.NewCoins(msg.Amount)); err != nil {
		return nil, err
	}

	lockMultiplier := types.GetLockMultiplier(msg.LockDuration)

	staking := &types.Staking{
		Id:             m.IncrementStakingId(ctx),
```

```
  Address:        msg.Staker,
  Amount:         msg.Amount,
  LockDuration:   msg.LockDuration,
  LockMultiplier: lockMultiplier,
  EffectiveAmount: types.GetEffectiveAmount(msg.Amount, lockMultiplier),
  PendingRewards: sdk.NewCoin(m.RewardPerEpoch(ctx).Denom, sdkmath.ZeroInt()),
  TotalRewards:   sdk.NewCoin(m.RewardPerEpoch(ctx).Denom, sdkmath.ZeroInt()),
  StartTime:      ctx.BlockTime(),
  Status:         types.StakingStatus_STAKING_STATUS_STAKED,
}

// set staking
m.SetStaking(ctx, staking)
m.SetStakingByAddress(ctx, msg.Staker, staking)

// update total staking
m.IncreaseTotalStaking(ctx, staking)

// emit events
ctx.EventManager().EmitEvent(
  sdk.NewEvent(
    types.EventTypeStake,
    sdk.NewAttribute(types.AttributeKeyStaker, msg.Staker),
    sdk.NewAttribute(types.AttributeKeyId, fmt.Sprintf("%d", staking.Id)),
    sdk.NewAttribute(types.AttributeKeyAmount, msg.Amount.String()),
    sdk.NewAttribute(types.AttributeKeyLockDuration, msg.LockDuration.String()),
  ),
)

return &types.MsgStakeResponse{}, nil
}
```

Listing 2.19: side/x/farming/keeper/msg_server.go

Impact Runtime panic due to the unrestricted staking requests.

Suggestion Revise the logic accordingly.

2.1.11 Potential runtime panic due to the improper update of `RewardPerEpoch`

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the file `msg_server.go` of the `farming` module, the function `Updateparams()` updates the parameters of the module `farming`. However, it lacks validation for the parameter `RewardPerEpoch`, potentially leading to runtime panic issues. Specifically, the reward coin (i.e., `RewardPerEpoch`) is updated when there are active stakings with non-zero pending rewards, the runtime panic occurs during accumulating the pending rewards for active stakings in the function `EndBlocker()`.

```
func (k Keeper) OnParamsChanged(ctx sdk.Context, params types.Params, newParams types.Params) {
```

```
if !params.Enabled && newParams.Enabled {
  // start the new epoch when farming enabled or re-enabled
  k.NewEpoch(ctx)
} else if params.Enabled && !newParams.Enabled {
  // remove the staking queue for the current epoch
  k.RemoveCurrentEpochStakingQueue(ctx)

  // end the current epoch
  currentEpoch := k.GetCurrentEpoch(ctx)
  currentEpoch.Status = types.EpochStatus_EPOCH_STATUS_ENDED
  k.SetEpoch(ctx, currentEpoch)
}
}
```

Listing 2.20: side/x/farming/keeper/params.go

Impact Runtime panic due to the improper update of the parameter `RewardPerEpoch`.

Suggestion Add checks when updating the parameter `RewardPerEpoch`.

2.1.12 Incorrect reward estimation

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the project, users' staking rewards are settled in epochs. However, in the function `NewEpoch()`, the new epoch's `StartTime` is set based on `BlockTime()` instead of the previous epoch's `EndTime`. This design creates gaps between two adjacent epochs. As a result, users may not earn rewards for the entire lock duration.

```
func (k Keeper) NewEpoch(ctx sdk.Context) {
 epoch := &types.Epoch{
   Id:        k.IncrementEpochId(ctx),
   StartTime: ctx.BlockTime(),
   EndTime:   ctx.BlockTime().Add(k.EpochDuration(ctx)),
   Status:    types.EpochStatus_EPOCH_STATUS_STARTED,
 }

 // set the new epoch
 k.SetEpoch(ctx, epoch)

 // call handler on the new epoch started
 k.OnEpochStarted(ctx)
}
```

Listing 2.21: side/x/farming/keeper/epoch.go

```
// OnEpochStarted is called when the current epoch is started
func (k Keeper) OnEpochStarted(ctx sdk.Context) {
 // get the current epoch
 currentEpoch := k.GetCurrentEpoch(ctx)

```

```go
// get staked stakings
stakings := k.GetStakingsByStatus(ctx, types.StakingStatus_STAKING_STATUS_STAKED)

for _, staking := range stakings {
  // ensure the staking end time satisfies the current epoch
  if staking.StartTime.Add(staking.LockDuration).Before(currentEpoch.EndTime) {
    continue
  }

  // add to staking queue for the current epoch
  k.AddToCurrentEpochStakingQueue(ctx, staking)

  // update total stakings for the current epoch
  types.UpdateEpochTotalStakings(currentEpoch, staking)
}

// update the current epoch
k.SetEpoch(ctx, currentEpoch)
}
```

Listing 2.22: side/x/farming/keeper/epoch.go

```go
// GetNextEpochSnapshot gets the current snapshot for the next epoch
func (k Keeper) GetNextEpochSnapshot(ctx sdk.Context) *types.Epoch {
// get the current epoch
currentEpoch := k.GetCurrentEpoch(ctx)

// next epoch
nextEpoch := &types.Epoch{
  StartTime: currentEpoch.EndTime,
  EndTime:   currentEpoch.EndTime.Add(k.EpochDuration(ctx)),
}

// get staked stakings
stakings := k.GetStakingsByStatus(ctx, types.StakingStatus_STAKING_STATUS_STAKED)

for _, staking := range stakings {
  // ensure the staking end time satisfies the next epoch
  if staking.StartTime.Add(staking.LockDuration).Before(nextEpoch.EndTime) {
    continue
  }

  // update total stakings for the next epoch
  types.UpdateEpochTotalStakings(nextEpoch, staking)
}

return nextEpoch
}
```

Listing 2.23: side/x/farming/keeper/epoch.go

Listing 2.24: side/x/farming/keeper/params.go

Moreover, in the file `reward.go`, the function `GetEstimatedReward()` estimates rewards for a user. This function invokes the function `GetNextEpochSnapshot()` to construct the next epoch by collecting potential stakings. However, the stakings collected for the next epoch are potentially invalid, due to the incorrect assignment for the filed `Epoch.StartTime`. Specifically, in the function `GetNextEpochSnapshot()`, the `Epoch.StartTime` is set to the previous epoch's `EndTime`, which differs from the real start time (i.e., `BlockTime()`) used in the function `NewEpoch()`. As a result, the function `GetNextEpochSnapshot()` may include more stakings, leading to incorrect reward estimation.

```go
func (k Keeper) GetEstimatedReward(ctx sdk.Context, address string, amount sdk.Coin, lockDuration
    time.Duration) *types.AccountRewardPerEpoch {
	nextEpoch := k.GetNextEpochSnapshot(ctx)

	staking := &types.Staking{
		Address:         address,
		Amount:          amount,
		EffectiveAmount: types.GetEffectiveAmount(amount, types.GetLockMultiplier(lockDuration)),
	}

	types.UpdateEpochTotalStakings(nextEpoch, staking)

	totalStakings := []types.TotalStaking{
		{
			Denom:           staking.Amount.Denom,
			Amount:          staking.Amount,
			EffectiveAmount: staking.EffectiveAmount,
		},
	}

	for _, staking := range k.GetStakingsByAddress(ctx, address) {
		if staking.Status == types.StakingStatus_STAKING_STATUS_STAKED && !staking.StartTime.Add(
		    staking.LockDuration).Before(nextEpoch.EndTime) {
			totalStakings = types.UpdateAccountTotalStakings(totalStakings, staking)
		}
	}

	return types.GetAccountRewardPerEpoch(address, totalStakings, nextEpoch, k.RewardPerEpoch(ctx),
	    k.EligibleAssets(ctx))
}
```

Listing 2.25: side/x/farming/keeper/reward.go

```go
// GetNextEpochSnapshot gets the current snapshot for the next epoch
func (k Keeper) GetNextEpochSnapshot(ctx sdk.Context) *types.Epoch {
	// get the current epoch
	currentEpoch := k.GetCurrentEpoch(ctx)

	// next epoch
	nextEpoch := &types.Epoch{
		StartTime: currentEpoch.EndTime,
		EndTime:   currentEpoch.EndTime.Add(k.EpochDuration(ctx)),
	}
```

```go

 // get staked stakings
 stakings := k.GetStakingsByStatus(ctx, types.StakingStatus_STAKING_STATUS_STAKED)

 for _, staking := range stakings {
   // ensure the staking end time satisfies the next epoch
   if staking.StartTime.Add(staking.LockDuration).Before(nextEpoch.EndTime) {
     continue
   }

   // update total stakings for the next epoch
   types.UpdateEpochTotalStakings(nextEpoch, staking)
 }

 return nextEpoch
}
```

Listing 2.26: side/x/farming/keeper/epoch.go

Impact Incorrect reward estimation in the function `GetEstimatedReward()`.

Suggestion Revise the logic accordingly.

2.1.13 Potential DoS in the DKG completion process

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the file `tss.go`, the function `CheckDKGCompletions()` verifies whether all participants have submitted completions and whether all `completions` contain identical `PubKeys`. However, in the file `msg_server.go`, the function `CompleteDKG()` allows a malicious participant to submit a completion with customized `pubKeys` and `signature` that can still pass the validations in the function `CompleteDKG()` of the file `dkg.go`. As a result, this design could lead to a DoS issue in the DKG completion process on the App chain due to the malicious completion. Additionally, a DoS risk also exists if completions are missing (e.g., some participants are unintentionally offline).

```go
func CheckDKGCompletions(completions []*DKGCompletion) bool {
 if len(completions) == 0 {
   return false
 }

 pubKeys := completions[0].PubKeys

 for _, completion := range completions[1:] {
   if !reflect.DeepEqual(completion.PubKeys, pubKeys) {
     return false
   }
 }

 return true
```

```
}
```

Listing 2.27: side/x/tss/types/tss.go

```go
func (m msgServer) CompleteDKG(goCtx context.Context, msg *types.MsgCompleteDKG) (*types.
    MsgCompleteDKGResponse, error) {
  if err := msg.ValidateBasic(); err != nil {
    return nil, err
  }

  ctx := sdk.UnwrapSDKContext(goCtx)

  if err := m.Keeper.CompleteDKG(ctx, msg.Sender, msg.Id, msg.PubKeys, msg.ConsensusPubkey, msg.
      Signature); err != nil {
    return nil, err
  }

  dkgRequest := m.GetDKGRequest(ctx, msg.Id)

  // callback to the module handler
  if err := m.GetDKGCompletionReceivedHandler(dkgRequest.Module)(ctx, dkgRequest.Id, dkgRequest.
      Type, dkgRequest.Intent, msg.ConsensusPubkey); err != nil {
    return nil, err
  }

  // Emit events
  ctx.EventManager().EmitEvent(
    sdk.NewEvent(
      types.EventTypeCompleteDKG,
      sdk.NewAttribute(types.AttributeKeySender, msg.Sender),
      sdk.NewAttribute(types.AttributeKeyId, fmt.Sprintf("%d", msg.Id)),
      sdk.NewAttribute(types.AttributeKeyParticipant, msg.ConsensusPubkey),
    ),
  )

  return &types.MsgCompleteDKGResponse{}, nil
}
```

Listing 2.28: side/x/tss/keeper/msg_server.go

```go
func (k Keeper) CompleteDKG(ctx sdk.Context, sender string, id uint64, pubKeys []string,
    consensusPubKey string, signature string) error {
  if !k.HasDKGRequest(ctx, id) {
    return types.ErrDKGRequestDoesNotExist
  }

  dkgRequest := k.GetDKGRequest(ctx, id)
  if dkgRequest.Status != types.DKGStatus_DKG_STATUS_PENDING {
    return errorsmod.Wrap(types.ErrInvalidDKGStatus, "dkg request non pending")
  }

  if !ctx.BlockTime().Before(dkgRequest.ExpirationTime) {
    return types.ErrDKGRequestExpired
```

```
}

if !types.ParticipantExists(dkgRequest.Participants, consensusPubKey) {
  return types.ErrUnauthorizedParticipant
}

if k.HasDKGCompletion(ctx, id, consensusPubKey) {
  return types.ErrDKGCompletionAlreadyExists
}

if len(pubKeys) != int(dkgRequest.BatchSize) {
  return errorsmod.Wrap(types.ErrInvalidDKGCompletion, "mismatched public key count")
}

if !types.VerifySignature(signature, consensusPubKey, types.GetDKGCompletionSigMsg(id, pubKeys)) {
  return types.ErrInvalidSignature
}

completion := &types.DKGCompletion{
  Id:              id,
  Sender:          sender,
  PubKeys:         pubKeys,
  ConsensusPubkey: consensusPubKey,
  Signature:       signature,
}

k.SetDKGCompletion(ctx, completion)

return nil
}
```

Listing 2.29: side/x/tss/keeper/dkg.go

Impact Potential DoS in the DKG completion process.

Suggestion Revise the logic accordingly.

Feedback from the project The project stated that its trust model assumes all participants are honest and active.

2.1.14 Improper design of disabling the farming module

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the file `params.go`, when the parameter update intends to disable the farming module (i.e., `params.Enabled` == `True` && `newParams.Enabled` == `False`), the existing stakings are removed from the current epoch (i.e., the invocation of the `RemoveCurrentEpochStakingQueue()` function) without settling rewards for these stakings. In this case, the rewards of stakings in the current epoch can not be settled.

```
} else if params.Enabled && !newParams.Enabled {
  // remove the staking queue for the current epoch
  k.RemoveCurrentEpochStakingQueue(ctx)

  // end the current epoch
  currentEpoch := k.GetCurrentEpoch(ctx)
  currentEpoch.Status = types.EpochStatus_EPOCH_STATUS_ENDED
  k.SetEpoch(ctx, currentEpoch)
}
```

Listing 2.30: side/x/farming/keeper/params.go

Impact The rewards of stakings in the current epoch can not be settled.

Suggestion Revise the logic accordingly.

2.1.15 Lack of patching the cosmos-sdk package

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the cosmos-sdk, there are few issues (e.g., link1 and link2) having been fixed in the versions v0.50.13 and v0.50.14. The currently used version is v0.50.12, which is vulnerable to these issues.

Impact The chain may halt due to the lack of patching the cosmos-sdk package.

Suggestion Upgrade the cosmos-sdk package to a secure version.

2.2 Recommendation

2.2.1 Review the incorrect formula annotation

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the file `pool.go`, the formula annotation for the function `UpdatePoolTranches()` is inconsistent with the code logic. In addition, the formula annotation is incorrect. It is recommended to review the incorrect formula annotation.

```
// UpdatePoolTranches updates total borrowed amount for each tranche at the beginning of each block
//
// Formula:
//
// borrow rate = borrowAPR / blocksPerYear * (1-reserve factor)
// borrowIndex_new = borrowIndex_old * (1+borrow rate)
// totalBorrowed_new = totalBorrowed_old * borrowIndex_new/borrowIndex_old
func (k Keeper) UpdatePoolTranches(ctx sdk.Context, pool *types.LendingPool) {
 // get blocks per year
 blocksPerYear := k.GetBlocksPerYear(ctx)
```

```
  for i, tranche := range pool.Tranches {
    trancheConfig, _ := types.GetTrancheConfig(pool.Config.Tranches, tranche.Maturity)

    borrowRatePerBlock := sdkmath.LegacyNewDec(int64(trancheConfig.BorrowAPR)).Quo(sdkmath.LegacyNewDec(1000)).Quo(sdkmath.LegacyNewDec(int64(blocksPerYear)))
    borrowIndexRatio := sdkmath.LegacyOneDec().Add(borrowRatePerBlock)

    reserveDelta := pool.Tranches[i].TotalBorrowed.ToLegacyDec().Mul(borrowRatePerBlock).MulInt(sdkmath.NewInt(int64(pool.Config.ReserveFactor))).QuoInt(sdkmath.NewInt(1000)).TruncateInt()

    pool.Tranches[i].BorrowIndex = pool.Tranches[i].BorrowIndex.Mul(borrowIndexRatio)
    pool.Tranches[i].TotalBorrowed = pool.Tranches[i].TotalBorrowed.ToLegacyDec().Mul(borrowIndexRatio).TruncateInt()

    pool.Tranches[i].TotalReserve = pool.Tranches[i].TotalReserve.Add(reserveDelta)
  }
}
```

Listing 2.31: side/x/lending/keeper/pool.go

Suggestion Review the incorrect formula annotation.

2.2.2 Revise the typos

Status Fixed in `Version 3`

Introduced by `Version 1`

Description The word `AttributeKeyMuturityTime` should be `AttributeKeyMaturityTime`.

```
    sdk.NewAttribute(types.AttributeKeyMuturityTime, fmt.Sprint(loan.MaturityTime)),
```

Listing 2.32: side/x/lending/keeper/msg_server_loan.go

Suggestion Revise the typos.

2.2.3 Add duplicate checks for `Maturity` when configuring the pool's tranches

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the file `lending.go`, the function `validatePoolTranches()` validates the pool `tranches` before assigning the provided tranches to a pool. Specifically, it performs an empty check but does not perform a duplicate check for the input `tranches`. It is recommended to add duplicate checks when configuring the pool's tranches.

```
func NewTranches(trancheConfigs []PoolTrancheConfig) []PoolTranche {
  tranches := make([]PoolTranche, len(trancheConfigs))

  for i, config := range trancheConfigs {
    tranches[i].Maturity = config.Maturity
    tranches[i].BorrowIndex = InitialBorrowIndex
```

```
  }

  return tranches
}
```

Listing 2.33: side/x/lending/types/lending.go

```
pool := &types.LendingPool{
  Id:           msg.Id,
  Supply:       sdk.NewCoin(msg.Config.LendingAsset.Denom, sdkmath.ZeroInt()),
  TotalSTokens: sdk.NewCoin(sTokenDenom, sdkmath.ZeroInt()),
  Tranches:     types.NewTranches(msg.Config.Tranches),
```

Listing 2.34: side/x/lending/keeper/msg_server_pool.go

```
func validatePoolTranches(tranches []PoolTrancheConfig) error {
  if len(tranches) == 0 {
    return errorsmod.Wrap(ErrInvalidPoolConfig, "tranches can not be empty")
  }

  for _, tranche := range tranches {
    if tranche.Maturity <= 0 {
      return errorsmod.Wrap(ErrInvalidPoolConfig, "maturity must be greater than 0")
    }

    if tranche.BorrowAPR == 0 || tranche.BorrowAPR >= 1000 {
      return errorsmod.Wrap(ErrInvalidPoolConfig, "borrow apr must be between (0, 1000)")
    }

    if tranche.MinMaturityFactor == 0 || tranche.MinMaturityFactor > 1000 {
      return errorsmod.Wrap(ErrInvalidPoolConfig, "min maturity factor must be between (0, 1000]")
    }
  }

  return nil
}
```

Listing 2.35: side/x/lending/types/lending.go

Suggestion Add duplicate checks when configuring pools' tranches.

2.2.4 Remove redundant code

Status Confirmed

Introduced by `Version 1`

Description There are several unused variables, events, functions. It is recommended to remove them for better code readability. Specifically, the following code should be removed or revised.

1. The function `DecreasePendingLendingEventCount()` can not be invoked when there are no available lending events. Therefore, in the `DecreasePendingLendingEventCount()` function, the `if` branch is redundant.

```
if count == 0 {
  return
}
```

Listing 2.36: side/x/dlc/keeper/event.go

Suggestion Remove the redundant code.

2.2.5 Refactor the fee fetching logic

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the file `abci.go`, the function `handleCompletedLiquidations()` gets the `feeRate` via the function `BtcBridgeKeeper().GetFeeRate()` to construct a settlement transaction. If the retrieved `feeRate` is 0, the function `handleCompletedLiquidations()` breaks directly. It is recommended to add a default fee for building the settlement transaction when the fetched `feeRate` is zero.

```
feeRate := k.BtcBridgeKeeper().GetFeeRate(ctx)
if err := k.BtcBridgeKeeper().CheckFeeRate(ctx, feeRate); err != nil {
  k.Logger(ctx).Info("Failed to get fee rate to handle liquidation", "err", err)

  return
}
```

Listing 2.37: side/x/liquidation/module/abci.go

Suggestion Refactor the `feeRate` fetching logic.

2.2.6 Perform proper cleanup in the function `Unstake()`

Status Confirmed

Introduced by `Version 2`

Description In the file `msg_server.go`, the function `Unstake()` does not invoke the function `SetStakingByAddress()` for unstaked stakings. Without removing these unstaked stakings, the workload increase for functions such as `ClaimAllRewards()` and `GetEstimatedReward()`, which iterate over all stakings of an address via the function `IterateStakingsByAddress()`. It is recommended to invoke the function `SetStakingByAddress()` in the function `Unstake()` to perform proper cleanup.

```
func (k Keeper) ClaimAllRewards(ctx sdk.Context, address string) (sdk.Coin, error) {
pendingRewards := sdk.NewCoin(k.RewardPerEpoch(ctx).Denom, sdkmath.ZeroInt())

k.IterateStakingsByAddress(ctx, address, func(staking *types.Staking) (stop bool) {
  // accumulate pending rewards
  pendingRewards = pendingRewards.Add(staking.PendingRewards)

  // reset pending rewards
  staking.PendingRewards = sdk.NewCoin(k.RewardPerEpoch(ctx).Denom, sdkmath.ZeroInt())
```

```
    k.SetStaking(ctx, staking)

    return false
  })

  if pendingRewards.IsZero() {
    return pendingRewards, types.ErrNoPendingRewards
  }

  if err := k.bankKeeper.SendCoinsFromModuleToAccount(ctx, types.ModuleName, sdk.
      MustAccAddressFromBech32(address), sdk.NewCoins(pendingRewards)); err != nil {
    return pendingRewards, err
  }

  return pendingRewards, nil
}
```

Listing 2.38: side/x/farming/keeper/reward.go

```
func (m msgServer) Unstake(goCtx context.Context, msg *types.MsgUnstake) (*types.
    MsgUnstakeResponse, error) {
  if err := msg.ValidateBasic(); err != nil {
    return nil, err
  }

  ctx := sdk.UnwrapSDKContext(goCtx)

  if !m.HasStaking(ctx, msg.Id) {
    return nil, errorsmod.Wrapf(types.ErrStakingDoesNotExist, "id: %d", msg.Id)
  }

  staking := m.GetStaking(ctx, msg.Id)
  if staking.Address != msg.Staker {
    return nil, errorsmod.Wrap(types.ErrUnauthorized, "mismatched staker address")
  }

  if staking.Status == types.StakingStatus_STAKING_STATUS_UNSTAKED {
    return nil, errorsmod.Wrapf(types.ErrInvalidStakingStatus, "already unstaked: %d", msg.Id)
  }

  if ctx.BlockTime().Before(staking.StartTime.Add(staking.LockDuration)) {
    return nil, errorsmod.Wrapf(types.ErrLockDurationNotEnded, "lock duration end time: %s",
        staking.StartTime.Add(staking.LockDuration))
  }

  if err := m.bankKeeper.SendCoinsFromModuleToAccount(ctx, types.ModuleName, sdk.
      MustAccAddressFromBech32(msg.Staker), sdk.NewCoins(staking.Amount)); err != nil {
    return nil, err
  }

  // claim pending rewards if any
  if staking.PendingRewards.IsPositive() {
    if err := m.bankKeeper.SendCoinsFromModuleToAccount(ctx, types.ModuleName, sdk.
```

```
        MustAccAddressFromBech32(msg.Staker), sdk.NewCoins(staking.PendingRewards)); err != nil {
      return nil, err
    }

    // reset pending rewards
    staking.PendingRewards = sdk.NewCoin(staking.PendingRewards.Denom, sdkmath.ZeroInt())
  }

  // update status
  staking.Status = types.StakingStatus_STAKING_STATUS_UNSTAKED
  m.SetStaking(ctx, staking)

  // update total staking
  m.DecreaseTotalStaking(ctx, staking)
```

Listing 2.39: side/x/farming/keeper/msg_server.go

Suggestion Perform proper cleanup in the function `Unstake()`.

2.3 Note

2.3.1 The design of the loan's `Authorizations` field

Introduced by `Version 1`

Description In the file `msg_server_loan.go`, the function `SubmitCets()` creates an authorization with deposit transactions and appends the created authorization to the list `loan.Authorizations` (Line 202 and 225). A borrower can only invoke the function `SubmitCets()` once to create an authorization for each loan due to the status mutation (Line 228) and the check (Line 159). However, the field `Authorizations` of the struct `Loan` is designed as a list, which indicates that each loan can have multiple authorizations. The project must ensure that the field `Authorizations` of the struct `Loan` are properly used.

```
142 func (m msgServer) SubmitCets(goCtx context.Context, msg *types.MsgSubmitCets) (*types.
      MsgSubmitCetsResponse, error) {
143   if err := msg.ValidateBasic(); err != nil {
144     return nil, err
145   }
146
147   ctx := sdk.UnwrapSDKContext(goCtx)
148
149   if !m.HasLoan(ctx, msg.LoanId) {
150     return nil, types.ErrLoanDoesNotExist
151   }
152
153   loan := m.GetLoan(ctx, msg.LoanId)
154   if msg.Borrower != loan.Borrower {
155     return nil, types.ErrMismatchedBorrower
156   }
157
158   // NOTE: only can be authorized once for now
159   if loan.Status != types.LoanStatus_Requested {
```

```
  return nil, errorsmod.Wrap(types.ErrInvalidLoanStatus, "loan non requested")
}
```

Listing 2.40: side/x/lending/keeper/msg_server_loan.go

```
authorization := m.CreateAuthorization(ctx, msg.LoanId, depositTxHashes)

for i, depositTx := range msg.DepositTxs {
  if !m.HasDepositLog(ctx, depositTxHashes[i]) {
    depositLog := &types.DepositLog{
      Txid:           depositTxHashes[i],
      VaultAddress:   loan.VaultAddress,
      AuthorizationId: authorization.Id,
      DepositTx:      depositTx,
    }

    m.SetDepositLog(ctx, depositLog)
  }
}

collateralDecimals := int(poolConfig.CollateralAsset.Decimals)
borrowDecimals := int(poolConfig.LendingAsset.Decimals)
collateralIsBaseAsset := poolConfig.CollateralAsset.IsBasePriceAsset

// calculate liquidation price
liquidationPrice := types.GetLiquidationPrice(collateralAmount, collateralDecimals, loan.
    BorrowAmount.Amount, borrowDecimals, loan.Maturity, loan.BorrowAPR, m.GetBlocksPerYear(ctx),
     poolConfig.LiquidationThreshold, collateralIsBaseAsset)

// update loan
loan.Authorizations = append(loan.Authorizations, *authorization)
loan.CollateralAmount = collateralAmount
loan.LiquidationPrice = liquidationPrice
loan.Status = types.LoanStatus_Authorized
m.SetLoan(ctx, loan)

return &types.MsgSubmitCetsResponse{}, nil
}
```

Listing 2.41: side/x/lending/keeper/msg_server_loan.go

Feedback from the project The project stated that the design is for the collateral addition, which is not available now. The project will refactor the design in the future.

2.3.2 The design of liquidation and bad debt management

Introduced by `Version 1`

Description During the liquidation process, the collateral is only released and distributed to liquidators when the debt is fully paid or the collateral is fully consumed. This design may hinder the collateral distribution for liquidators if the liquidation process takes a long time. Moreover, when the bad debt exists due to the sharp price fall of the collateral, the liquidation may be hindered, leading to a delay of the collateral distribution. As a result, liquidators can not retrieve their liquidated collateral instantly, causing a potential loss. The project must ensure a smooth liquidation process by effectively facilitating liquidations and appropriately managing bad debt.

Feedback from the project The project acknowledges this issue and plans to optimize the liquidation process in future iterations.

2.3.3 The design of price queries

Introduced by `Version 1`

Description The `URL`s used for the price service are hardcoded. A failure in the price service may result in incorrect system behavior. The project must ensure the price service is working properly or the price service failures are properly handled properly (e.g., via a timely software upgrade).

```go
var (
  ProviderName = "binance"
  SymbolMap   = map[string]string{
    "BTCUSDT": types.BTCUSD,
  }
  URL         = "wss://stream.binance.com:443/stream?streams=btcusdt@miniTicker/ethbtc@miniTicker"
  SubscribeMsg = ""
)

func symbol(source string) string {
  if target, ok := SymbolMap[source]; ok {
    return target
  } else {
    return source
  }
}

func Subscribe(svrCtx *server.Context, ctx context.Context) error {
  return types.Subscribe(ProviderName, svrCtx, ctx, URL, SubscribeMsg, func(msg []byte) []types.Price {
    subscription := &Subscription{}
    prices := []types.Price{}
    if err := json.Unmarshal(msg, &subscription); err == nil {
      price := types.Price{
        Symbol: symbol(subscription.Data.Symbol),
        Price:  subscription.Data.Close,
        Time:   subscription.Data.EventTime,
      }
      prices = append(prices, price)
    }
    return prices
  })
}
```

Listing 2.42: side/x/oracle/providers/binance/provider.go

2.3.4 Potential centralization risks

Introduced by `Version 1`

Description The execution result of the Bitway App Chain is determined by the validators. The validators should be properly distributed and decentralized to ensure the security and trustworthiness of the protocol. If the validators are concentrated in the hands of a few entities or lack proper geographical and organizational distribution, it could introduce potential centralization risks.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS