ANCILIA

Sumer AUDITING REPORT

v1.6

Aug 2024

Prepared for

Sumer.money

Prepared by

Ancilia, Inc.

Index

Version History

Version	Description	Date
v1.6	Minor update	08/29/2024
v1.5	Compact complogic and improved Oracle	08/22/2024
v1.4	Add Oracle audit findings	07/15/2024
v1.3	Reviewed one more branch	06/21/2024
v1.2	Final report	06/18/2024
v1.1	Add new findings	06/03/2024
v1.0	Initial report	05/20/2024

Executive Summary

The Sumer.money team shared their smart contract source code via github. We have listed tags of smart contracts to ensure the entirety of the audit can be tied to a given contract version. The Ancilia team has worked with the Sumer.money team on all potential findings and issues. The audit scope includes checking for smart contracts with attack vulnerabilities such as re-entry attacks, logic flaws, authentication bypasses, DoS attacks, etc. Our researchers primarily focused on CToken, Oracle, Comptroller and interest models. The deployment scripts, staking and underlying token contracts are not in the scope of this audit.
The original Sumer-project is forked from Compound protocol. Any logic within their original code is not in the scope of this audit.

Disclaimer

Note that security audit services do not guarantee to find all possible security issues in the given smart contracts. A repeating code audit or incremental code audit is encouraged. Multiple audits with several auditors are recommended. Product owners are still required to have their own test cases and regular code review process. A threat intelligence system may help to discover or prevent a potential attack which can further reduce risk. Additionally, a bug bounty program for the community will help improve the security of products. Last but not least, Security is complicated! A strong smart contract does not guarantee your product is safe from all cybersecurity attacks.

Contracts overview

The contracts repository was shared through github, the tag names and commit revisions are attached below. The last tag we have audit is "refs/tags/audit-final".

Repo	Tags	Revision
sumer-project	refs/tags/audit	32cd5d87079c64e68efc8a2faf4f427137db3e62
	refs/tags/audit-fix1	0319c31c2703548484886492d8612f2325cdd541
	refs/tags/audit-fix2	bf952f2e726dfe151b3f65aa86b0e0ec96cd8993
	refs/tags/audit-fix3	f59a83e0cebd69c8130d16dbe7691ebe4d2d46e8
	refs/tags/audit2	dd885c4d607da0ea1d7b3d1fac4332752ef64c78
	refs/tags/audit2-fix1	1280aebcc45033eb7e8825d15c282d53fc3cf6c5
	refs/tags/audit2-fix2	9492ad9131fddf4fcec585cf286ab1d930e85287
	refs/tags/audit2-fix3	370d151c12287babc6fca82f8d37a3e9724ca1ab
	refs/tags/audit2-fix4	7216cc4329d8d0e837ded83c8df7e9c8e9325b72
	refs/tags/audit2-fix5	08a0e01502b5549a77bea13ff6ae7be1bee967d4
	refs/tags/audit2-fix6	f20ad32f2341f7a571fdaaa01c65812417fc73d5
	refs/tags/audit2-fix7	c268541360b81a0fe361f623054a75b104f67a64
	refs/tags/audit2-final	f950102bbf860a1eab4b9bac4e710db4f8dd3716
	refs/tags/audit2-final2	c3597c6a1f6749da06e69487a0d9f2734e561bb3
	refs/tags/v2-deployed	5c48ba95cdc3f5a1f2bd9eb512e03a8aabb91167
	refs/tags/audit-final	7f7754f238a13f79c7bacd3cc993d159454c394a
	refs/tags/audit-final-dev1	c7c86a37317e79cba09c3e0e9114c1e4528a22c9
	refs/tags/audit-intermint-fix2	eaabd8571dedae6d5f2641adedd76a1ac32d240e
	refs/tags/audit-by-ancilia	2c279836fef88a1a6e82401eb80b6a11fcd146c2
sumer-oracle	master	be22c2bd4f565e6db0b506ac4c3e49a5c75ab73f

The findings

Results

ID	Description	Severity	Product Impact	Status

Sumer-A-01	Arbitrary borrow and collateral for any account	Critical	Critical	Fixed
Sumer-A-02	The assetGroup index starts from 0	Medium	High	Fixed
Sumer-A-03	force to sync balance may lead to donation attack	Medium	Medium	Acked
Sumer-A-04	Unprotected privilege function	Critical	Critical	Fixed
Sumer-A-05	SortedBorrower on arbitrary token and borrower	High	High	Deprecated
Sumer-A-06	incorrect msg.sender after an external call	Medium	Medium	Fixed
Sumer-A-07	Native token address is inconsistent	Medium	Medium	Fixed
Sumer-A-08	Not following Checks-effects-interactions best practices	Medium	Medium	Fixed
Sumer-A-09	High gas used when updating the borrower	Medium	Medium	Deprecated
Sumer-A-10	Multiple Agreement Indexes might not work as expected	Medium	Medium	Fixed
Sumer-A-11	Timelock admin functions parameter checks	Low	Low	Fixed
Sumer-A-12	utilizationRate() may not be safe in WhitePaperInterestRateModel	High	Low	Fixed
Sumer-A-13	sendValue() does not have gas limits	Low	Low	Fixed
Sumer-A-14	New redeemFaceValue() function needs to verify deadline and chain ID	Low	Low	Fixed
Sumer-A-15	consumeValue() allowed to consume any CToken amount	Critical	Critical	Fixed
Sumer-A-16	createAgreement() can drain the pool	Critical	Critical	Fixed
Sumer-A-17	Oracle has Valid price for any arbitrary token contracts	Critical	Critical	Fixed
Sumer-A-18	High Gas consumption for Pendle Oracle	Low	Low	Improved

Sumer-A-19	The getGroupSummary() may miscalculate the borrowing debt	Critical	Critical	Fixed
Sumer-A-20	Token Decimals normalization concern	Info	Info	Testing Tool
Sumer-A-21	Inconsistency account liquidity check	Critical	Critical	Fixed
Sumer-A-22	GroupId could be set to invalid	Low	Low	Fixed
Sumer-A-23	Arbitrary largestGroupId setting could cut-off groups	Info	Info	Off chain
Sumer-A-24	Ambitious event when update an asset group setting	Low	Low	Fixed
Sumer-A-25	SupplyCap check logic flaw	Low	Low	Fixed
Sumer-A-26	maxClose calculation logic flow	Medium	Medium	Fixed
Sumer-A-27	Initialized state could be reset after upgrade	Critical	Critical	Fixed

Details

Sumer-A-01 [Critical] Arbitrary borrow and collateral for any account

In the contract CToken.sol, the "isCToken()" check in the function borrowAndDepositBack() can be bypassed. This allows anyone to add borrow/collateral on behalf of any account.

```
ftrace | funcSig
1398  function borrowAndDepositBack(address borrower⬆, uint256 borrowAmount⬆) external nonReentrant returns (uint256) {
1399    // only allowed to be called from su token
1400    if (CToken(msg.sender).isCToken()) {
1401      revert NotSuToken();
1402    }
1403    // only cToken has this function
1404    if (!isCToken) {
1405      revert NotCToken();
1406    }
1407    return borrowAndDepositBackInternal(payable(borrower⬆), borrowAmount⬆);
1408  }
1409
1410  /**
1411   * @notice Sender borrows assets from the protocol and deposit all of them back to the protocol
1412   * @param borrowAmount The amount of the underlying asset to borrow and deposit
1413   * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
1414   */
ftrace | funcSig
1415  function borrowAndDepositBackInternal(address payable borrower⬆, uint256 borrowAmount⬆) internal returns (uint256) {
1416    accrueInterest();
1417    borrowFresh(borrower⬆, borrowAmount⬆, false);
1418    mintFresh(borrower⬆, borrowAmount⬆, false);
1419    return uint256(0);
1420  }
1421 }
```

Suggestion: Only allow suToken to call this function.

Update: Fixed

Sumer-A-04 [critical] Unprotected privileged function

In the contract Comptroller.sol, the privileged function cleanAssetGroup() does not have any protection which allows anyone to reset(delete) all assetGroups. The missing assetGroup could impact the user's liquidity check.

```
593  function cleanAssetGroup() external {
594    for (uint8 i = 0; i < _eqAssetGroups.length; i++) {
595      uint8 groupId = _eqAssetGroups[i].groupId;
596      delete assetGroupIdToIndex[groupId];
597    }
598
599    uint8 len = uint8(_eqAssetGroups.length);
600    for (uint8 i = 0; i < len; i++) {
601      _eqAssetGroups.pop();
602    }
603  }
```

Suggestion: Allows only admins to call this function.

Update: Only allows calls from DEFAULT_ADMIN_ROLE role.

Sumer-A-15 [Critical] consumeValue() allowed to consume any CToken amount

In the contract Timelock.sol, the onlyListedCToken() modifier for the function consumeValue() ensures the cToken parameter is a valid token. However, there is no permission check on who can call this function. It allows anyone to consume any amount of underlying tokens on any valid cToken.

```
58  function consumeValue(uint256 underlyingAmount, address cToken) external onlyListedCToken(cToken) {
59      consumeValueInternal(underlyingAmount, cToken);
60  }
61
```

Suggestion: Check msg.sender to ensure it is authorized.
Update: Only allows msg.sender is a valid CToken.

Sumer-A-16 [Critical] createAgreement() can drain the pool

In the contract Timelock.sol, the onlyListedCToken() modifier for the function createAgreement() ensures the cToken parameter is a valid token. However, there is no permission check on who could call this function. It allows anybody to create a timelock agreement for any amount of underlying tokens on any valid cToken.

```
136 function createAgreement(
137     TimeLockActionType actionType,
138     address cToken,
139     uint256 underlyingAmount,
140     address beneficiary
141 ) external onlyListedCToken(cToken) returns (uint256) {
142     require(beneficiary != address(0), 'Beneficiary cant be zero address');
143     uint256 underlyBalance;
144     address underlying = ICToken(cToken).underlying();
145     if (underlying == address(0)) {
146         underlyBalance = address(this).balance;
147     } else {
148         underlyBalance = IERC20(underlying).balanceOf(address(this));
149     }
150     require(underlyBalance >= balances[underlying] + underlyAmount, 'balance error');
151     balances[underlying] = underlyBalance;
152
153     uint256 agreementId = agreementCount++;
154     uint48 timestamp = uint48(block.timestamp);
```

Suggestion: Check msg.sender to ensure it is authorized.
Update: Only allows msg.sender is a valid CToken.

Sumer-A-17 [Critical] Oracle has Valid price for any arbitrary token contracts

In the contract ResilientOracle, the function _getUnderlyingAsset() does not check if the CToken address is listed or not. Any contracts who implemented isCEther() and underlying() will have a valid price for the Oracle. It creates the opportunity that the Compound protocol may get a fake CToken price and causes uncertainty issues.

```
453     function _getUnderlyingAsset(address vToken⬆) private view notNullAddress(vToken⬆) returns (address
            asset) {
454         if (VBep20Interface(vToken⬆).isCEther()) {
455             asset = nativeAsset;
456         } else {
457             asset = VBep20Interface(vToken⬆).underlying();
458         }
459     }
460 }
```

```
ftrace | funcSig
1051    function getUnderlyingPriceNormalized(address cToken⬆) public view returns (uint256) {
1052      uint256 priceMantissa = oracle.getUnderlyingPrice(cToken⬆);
1053      if (priceMantissa <= 0) {
1054        revert PriceError();
1055      }
1056      uint decimals = ICToken(cToken⬆).decimals();
1057      if (decimals < 18) {
1058        priceMantissa = priceMantissa * (10 ** (18 - decimals));
1059      }
1060      return priceMantissa;
1061    }
1062
```

Suggestion: Check cToken if it is listed in Comptroller before calling oracle.getUnderlyingPrice().

Update: Fixed

Sumer-A-19 [Critical] The getGroupSummary() may miscalculate the borrowing debt

In the contract AccountLiquidity, the function getGroupSummary() does not add up the borrowing token amount which could lead to an arbitrary borrow amount on any tokens.

```
77      (, uint8 assetGroupId, ) = comptroller.markets(asset);
78      (uint256 oErr, uint256 depositBalance, uint256 borrowBalance, uint256 exchangeRate
79        .getAccountSnapshot(account);
80      require(oErr == 0, 'snapshot error');
81
82      // skip the calculation to save gas
83      if (depositBalance == 0 && borrowBalance == 0) {
84        continue;
85      }
86
```

Suggestion: Check if the asset token is the borrowing token.
Update: Fixed

Sumer-A-27 [Critical] Initialized state could be reset after upgrade

In the contract CToken.sol, there are two new variables that have been introduced with new changes. However, the new variables will occupy the storage slot used by the old implementation contract, the two important variables are "initialized" and "initializing". It means the contract will no longer be initialized after contract upgrade. It potentially cause project to be compromised.

```
84    uint256 public underlyingBalance;
85
86    uint256 public constant percentScale = 1e14;
87
88    uint256 public totalSecures;
89
90    uint256 public override accrualBlockTimestamp;
91  }
92
```

Suggestion: re-initialize the contract after upgrade.
Update: Fixed

Sumer-A-21 [Critical] Inconsistency Account Liquidity Check

In the contract Comptroller.sol, the new function getHypotheticalAccountLiquidity() which is moved from old contract AccountLiquidity.sol(function getGroupSummary) has an inconsistency liquidity calculation issue. In certain situations the user who has SU token may be forced to liquidate.

```
939     if (g.groupId == targetGroupId) {
940       targetGroup = g;
941     } else {
942       if (targetIsSuToken) {
943         sumLiquidity += (g.interMintRate * g.cDepositVal) / expScale;
944       } else {
945         sumLiquidity += (g.interCRate * g.cDepositVal) / expScale;
946       }
947       sumLiquidity += (g.interSuRate * g.suDepositVal) / expScale;
948       sumBorrowPlusEffects = sumBorrowPlusEffects + g.cBorrowVal + g.suBorrowVal;
949     }
950   }
```

The inconsistency about the interCRate and interMintRate will cause the inconsistency liquidity calculation.

Suggestion: User liquidity check algorithm should be consistent.
Update: Fixed.

Sumer-A-02 [High] The assetGroup index starts from 0

In the contract Comptroller.sol, an index(assetGroupIdToIndex) is used to manage the asset group. However, the valid index starts from '0', which means an non-existent groupId will be valid. For example, an arbitrary groupId in the function removeAssetGroup() will cause the group index 0 to be removed.

```
ftrace | funcSig
579 function removeAssetGroup(uint8 groupId⬆) external onlyRole(DEFAULT_ADMIN_ROLE) returns (uint256) {
580   uint8 length = uint8(_eqAssetGroups.length);
581   uint8 lastGroupId = _eqAssetGroups[length - 1].groupId;
582   uint8 index = assetGroupIdToIndex[groupId⬆];
583
584   _eqAssetGroups[index] = _eqAssetGroups[length - 1];
585   assetGroupIdToIndex[lastGroupId] = index;
586   _eqAssetGroups.pop();
587   delete assetGroupIdToIndex[groupId⬆];
588
589   emit RemoveAssetGroup(groupId⬆, length);
590   return uint256(0);
591 }
592
```

Suggestion: Start the group index from 1, use 0 to differentiate an unexistant groupId.

Update: Fixed

Sumer-A-05 [High] SortedBorrower on arbitrary token and borrower

In the contract Comptroller.sol, a couple of functions are intended to be protected by the "onlyCToken()" modifier to prevent public calls. However, the check is not robust enough. It allows anyone to call redemptionManager.updateSortedBorrows() on an arbitrary Token and borrower address, which could corrupt the sortedBorrows storage. Thus, could be used to target redemption.

```
660  modifier onlyCToken() {
661      require(isContract(msg.sender), 'only ctoken');
662      ICToken(msg.sender).isCToken();
663      _;
664  }
665
```

```
1282 function borrowVerify(address cToken⬆, address borrower⬆, uint256
       borrowAmount⬆) external onlyCToken {
1283     // Shh - currently unused
1284     cToken⬆;
1285     borrower⬆;
1286     borrowAmount⬆;
1287     redemptionManager.updateSortedBorrows(cToken⬆, borrower⬆);
1288 }
1289
```

Suggestion: Update onlyCToken modifier to ensure it has a strong and robust check.

Update: Took out the code and used pass in providers for redemption.

Sumer-A-12 [High] utilizationRate() may not be safe in WhitePaperInterestRateModel

In the contract WhitePaperInterestRateModel.sol, the function utilizationRate() may not be safe if the cash is less than the reserves. This contract does not appear to be used anywhere, but please be aware of the potential vulnerability.

```
55  function utilizationRate(
56    uint256 cash,
57    uint256 borrows,
58    uint256 reserves
59  ) public pure returns (uint256) {
60    // Utilization rate is 0 when there are no borrows
61    if (borrows == 0) {
62      return 0;
63    }
64
65    return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
66  }
```

Suggestion: Don't use this code without updating interest rate model
Update: Fixed

Sumer-A-08 [Medium] Not following Checks-effects-interactions best practice

To prevent potential re-entrance issues, developers must follow the Checks-effects-interactions pattern. There are several places in which the state will change after an external call to the user managed address.

```
172  function doTransferOut(address payable to, uint256 amount) internal override {
173    /* Send the Ether, with minimal gas and revert on failure */
174    // to.transfer(amount);
175    (bool success, ) = to.call{gas: 5300, value: amount}('');
176    require(success, 'unable to send value, recipient may have reverted');
177    underlyingBalance -= amount;
178  }
```

```
156  function claim(uint256[] calldata agreementIndexes⬆) external nonReentrant {
157    uint256[] memory sorted = sort_array(agreementIndexes⬆);
158    require(!frozen, 'timeLock frozen');
159
160    for (uint256 i = 0; i < agreementIndexes⬆.length; i++) {
161      Agreement memory agreement = _validateAndDeleteAgreement(msg.sender, sorted[i]);
162      if (agreement.underlying == address(1)) {
163        // payable(agreement.beneficiary).transfer(agreement.amount);
164        Address.sendValue(payable(msg.sender), agreement.amount);
165      } else {
166        IERC20(agreement.underlying).safeTransfer(msg.sender, agreement.amount);
167      }
168      underlyingDetail[agreement.underlying].totalBalance -= agreement.amount;
169    }
170  }
```

```
261  function doTransferOut(address payable to⬆, uint256 amount⬆) internal virtual overr
262    ICToken token = ICToken(underlying);
263    token.transfer(to⬆, amount⬆);
264    underlyingBalance -= amount⬆;
265
266    bool success;
```

Suggestion: Update state before making the call.

Update: Fixed.

Sumer-A-03 [Medium] force to sync balance may lead to donation attack

In the CToken.sol, the function _syncUnderlyingBalance() will force updating the cash(underlyingBalance) with the existing balance in the token contract. Depending on when you call this function, it may lead to a donation attack which is a known existing issue for compound protocols.

Suggestion: Please be cautious when using this call.

Update: "Yes, totally understand. We called it only to fix the bug we had previously that cEther did not track the underlying balance properly"

Sumer-A-06 [Medium]incorrect msg.sender after an external call

In the contract Comptroller.sol, the function redeemFaceValueWithPermit() calls this.redeemFaceValue(). Because of the external call, the msg.sender in

function redeemFaceValue() will be changed to the address of Comptroller. This is no longer the original EOA address which calls redeemFaceValueWithPermit().

```
894  function redeemFaceValueWithPermit(
895    address suToken⬆,
896    uint256 amount⬆,
897    uint256 deadline⬆,
898    bytes memory signature⬆
899  ) external {
900    address underlying = ICToken(suToken⬆).underlying();
901    IEIP712(underlying).permit(msg.sender, suToken⬆, amount⬆, deadline⬆, signature⬆)
902    return this.redeemFaceValue(suToken⬆, amount⬆);
903  }
```

Suggestion: Update function redeemFaceValue() to 'public' and remove the external call by using 'this.'

Update: Fixed

Sumer-A-07 [Medium] Native token address is inconsistent

In the CEther contract, createAgreement() will use the current underlying address which is address(0). But in the timelock contract, the claim() function uses address(1) as the native token. This inconsistency may cause unexpected side effects.

Suggestion: Use the same address for the CEther underlying address.

Update: Fixed

Sumer-A-09 [Medium] High gas used when updating the borrower

The operation on the SortedBorrower storage is quite expensive. For example, the function updateSortedBorrows() will loop the list and find the closest address, then update prevId and nextId. It will load the storage from 'cold' and the gas cost is 2,100. For example, if there are 500 borrowers and need to insert to the end, the minimum gas cost would be 500 * 2 * 2100 = 2,100,000.

Suggestion: Please review the business logic to determine if this cost is necessary.

Update: Deprecated

Sumer-A-10 [Medium] Multiple Agreement Indexes might not work as expected

In the contract Timelock.sol, the function claim() takes a list of indexes for the agreement claim. But the agreement positioning might be changed during the function _validateAndDeleteAgreement() as the last position gets moved to the claimed one. If the index list contains an index which is the last one claimed then this no longer works as expected.

```
135  function _validateAndDeleteAgreement(
136    address beneficiary⬆,
137    uint256 agreementIndex⬆
138  ) internal returns (Agreement memory) {
139    uint256 length = uint256(userAgreements[beneficiary⬆].length);
140    require(agreementIndex⬆ < length, 'agreement index out of bound');
141    Agreement memory agreement = userAgreements[beneficiary⬆][agreementIndex⬆];
142    require(block.timestamp >= agreement.releaseTime, 'release time not reached');
143    require(!agreement.isFrozen, 'agreement frozen');
144
145    // Move the last element to the deleted spot.
146    // Remove the last element.
147    delete userAgreements[beneficiary⬆][agreementIndex⬆];
148    userAgreements[beneficiary⬆][agreementIndex⬆] = userAgreements[beneficiary⬆][userAgreements[beneficiary⬆].length -
149    userAgreements[beneficiary⬆].pop();
150
151    emit AgreementClaimed(beneficiary⬆, agreementIndex⬆, agreement.underlying, agreement.actionType, agreement.amount);
152
153    return agreement;
154  }
155
```

Suggestion: Don't support multiple agreements or combine together by using the address.

Update: Fixed

Sumer-A-26 [Medium] maxClose calculation logic flow

In the contract Comptroller.sol, the function liquidateBorrowAllowed() will ensure the repayAmount is in a range by comparing with maxClose. However the calculation of maxClose has a logic error. The variable percentScale(which is 1e14) has been multiplied twice.

```
774     uint256 priceMantissa = getUnderlyingPriceNormalized(cTokenBorrowed↑);
775     /* The liquidator may not repay more than what is allowed by the closeFactor */
776     uint256 maxClose = (uint256(globalConfig.closeFactorPercent) * percentScale * percentScale * borrowBalance) /
777       expScale;
778     uint256 maxCloseValue = (priceMantissa * maxClose) / expScale / expScale;
779     if (maxCloseValue < globalConfig.minCloseValue) {
780       if (repayAmount↑ > borrowBalance) {
781         revert TooMuchRepay();
782       }
783     } else {
784       if (repayAmount↑ > maxClose) {
785         revert TooMuchRepay();
```

Suggestion: Please double check if it needs multiple two times..
Update: Fixed

Sumer-A-11 [Low] Timelock admin functions parameter checks

In the contract Timelock.sol, there are multiple privilege functions(onlyAdmin) that do not have parameter value checks. Adding a value check will prevent future damage by incidental calls. For example, the lockDuration can be set to 0 in the setLockDuration function, if this is never intended to be set to said value, it should be checked.

```
80   function setLockDuration(address underlying↑, uint48 lockDuration↑) external onlyAdmin {
81     underlyingDetail[underlying↑].lockDuration = lockDuration↑;
82   }
83
```

Suggestion: Please double check the value range for important functions.

Update: Use a new function isAgreementMature() to check the lock time.

Sumer-A-13 [Low] sendValue() does not have gas limits

In the contract Timelock.sol, the function claim() will call the Address.sendValue() function to send native tokens. There are no gas limits which can introduce the potential re-entry vulnerabilities.

```
ftrace | funcSig
function claim(uint256[] calldata agreementIds⬆) external nonReentrant {
  require(!frozen, 'TimeLock is frozen');

  for (uint256 index = 0; index < agreementIds⬆.length; index++) {
    Agreement memory agreement = _validateAndDeleteAgreement(agreementIds⬆[index]);
    address underlying = ICToken(agreement.cToken).underlying();
    if (underlying == address(0)) {
      // payable(agreement.beneficiary).transfer(agreement.amount);
      Address.sendValue(payable(agreement.beneficiary), agreement.underlyAmount);
    } else {
      IERC20(underlying).safeTransfer(agreement.beneficiary, agreement.underlyAmount);
    }
    balances[underlying] -= agreement.underlyAmount;
  }
}
```

Suggestion: Add gas limits, similar to other places in the code base.
Update: Fixed

Sumer-A-14 [Low] New redeemFaceValue() function needs to verify deadline and chain ID

In the contract RedemptionManager.sol, the new function redeemFaceValue() will verify the pass in providers list. However, it does not check if the deadline parameter is expired or not. Furthermore, the chain ID should be part of signature verification data so that users cannot send to signing providers across chains.

```
ftrace | funcSig
function redeemFaceValue(
  address csuToken⬆,
  uint256 amount⬆,
  address[] memory providers⬆,
  uint256 deadline⬆,
  bytes memory signature⬆
) public {
  if (ICToken(csuToken⬆).isCToken() || !comptroller.isListed(csuToken⬆)) {
    revert InvalidSuToken();
  }

  if (signature⬆.length != 65) {
    revert InvalidSignatureLength();
  }
  bytes32 hash = keccak256(abi.encodePacked(deadline⬆, providers⬆));
```

Suggestion: Check deadline and add chain ID into hash function.
Update: Fixed

Sumer-A-18 [Low] High Gas consumption for Pendle Oracle

In the contract PendleOracle, the function getPrice() will call PtOracle.getPtToSyRate() and ResilientOracle.getPrice() which has so many underlying calls. The gas consumption is pretty high(consumed about 368k gas for one price query).

```
    function getPrice(address asset⬆) public view virtual returns (uint256) {
        TokenConfig memory tokenConfig = tokenConfigs[asset⬆];
        if (tokenConfig.asset != asset⬆) revert("unknown token");
        uint256 rate = underlyingPtOracle.getPtToSyRate(tokenConfig.market, tokenConfig.twapDuration);

        (IStandardizedYield sy, , ) = IPMarket(tokenConfig.market).readTokens();
        return (intermediateOracle.getPrice(sy.yieldToken()) * rate) / EXP_SCALE;
    }
}
```

Suggestion: Save unnecessary calls
Update: Improved. Down from 368k to 305k.

Sumer-A-22 [Low] GroupId could be set to invalid

In the contract Comptroller.sol, the new function _changeGroupIdForAsset() does not check if the newGroupId if is 0. If newGroupId has been set to 0, then the market group id will be 0 and it means a invalid group.

```
298  function _changeGroupIdForAsset(
299      address cToken⬆,
300      uint8 newGroupId⬆
301  ) external onlyRole(DEFAULT_ADMIN_ROLE) onlyListedCToken(cToken⬆) {
302      CompactAssetGroup memory g = assetGroup[newGroupId⬆];
303      if (g.groupId != newGroupId⬆) {
304          revert InvalidGroupId();
305      }
306      markets[cToken⬆].assetGroupId = newGroupId⬆;
307  }
308
```

Suggestion: Check if newGroupId if is 0.
Update: Fixed

Sumer-A-24 [Low] Ambitious event when update an asset group setting

In the contract Comptroller.sol, the new function setAssetGroup() allows an asset group config to be updated. But it still emits a "NewAssetGroup" event no matter what the action is.

```
412  function setAssetGroup(
413    uint8 groupId,
414    uint16 intraCRatePercent, // ctoken collateral rate for intra group ctoken liability
415    uint16 intraMintRatePercent, // ctoken collateral rate for intra group sutoken liability
416    uint16 intraSuRatePercent, // sutoken collateral rate for intra group ctoken liability
417    uint16 interCRatePercent, // ctoken collateral rate for inter group ctoken/sutoken liability
418    uint16 interSuRatePercent // sutoken collateral rate for inter group ctoken/sutoken liability
419  ) external onlyRole(DEFAULT_ADMIN_ROLE) {
420    assetGroup[groupId] = CompactAssetGroup(
421      groupId,
422      intraCRatePercent,
423      intraMintRatePercent,
424      intraSuRatePercent,
425      interCRatePercent,
426      interSuRatePercent
427    );
428    emit NewAssetGroup(
429      groupId,
430      intraCRatePercent,
431      intraMintRatePercent,
432      intraSuRatePercent,
433      interCRatePercent,
434      interSuRatePercent
435    );
436    if (groupId > globalConfig.largestGroupId) {
437      globalConfig.largestGroupId = groupId;
438    }
439  }
```

Suggestion: Better to differentiate the 'update' and 'new' actions.

Update: Fixed

Sumer-A-25 [Low] SupplyCap check logic flaw

In the contract Comptroller.sol, the function mintAllowed() will ensure the totalSupply won't over the supplyCap, however the calculation is incorrect.

```
530    uint256 exchangeRateMantissa = ICToken(cToken).exchangeRateStored();
531    if (
532      ((ICToken(cToken).totalSupply() + mintAmount) * exchangeRateMantissa) / expScale > marketConfig[cToken].supplyCap
533    ) {
534      revert SupplyCapReached();
535    }
536  }
```

Suggestion: mintAmount does not need to be converted by exchangeRate.

Update: Fixed

Sumer-A-20 [Info] Token Decimals normalization concern

When the token price has been normalized, it expects the Oracle returns token decimals 18. But not every Oracle returns the price in the 18 decimals. A necessary automation testing before adding assets to the market is required. Any decimal confusion will have a huge impact on the project.
Suggestion: Adding test script for every oracle and assets to ensure the no decimal issues.
Update:

Sumer-A-23 [Info] Arbitrary largestGroupId setting could cut-off Groups

In the contract Comptroller.sol, the new function _setLargestGroupId() will reset the largestGroupId to an arbitrary number which might cut off the group numbers in account liquidity function.

```
363  function _setLargestGroupId(uint8 largestGroupId⬆) external onlyRole(DEFAULT_ADMIN_ROLE) {
364      globalConfig.largestGroupId = largestGroupId⬆;
365  }
```

Suggestion: Check if the new largestGroupId value must not be less than any existing group id.
Update: Will check if off-chain.

Summary

Ancilia team has performed both an automated and manual code audit on the Sumer smart contracts mentioned above. All issues have been shared with the Sumer.money team through a telegram channel before this report. Overall, 8 critical, 3 high, 7 medium, 7 low and 2 informational impact issues have been discovered through this audit.
Sumer.money team reacted pretty quickly and fixed all the issues. Ancilia team verified and confirmed the fixes are in the github.