Sumer Project AUDITING REPORT

v1.4

July 2025

by **Tikkala Security**

Index

Update History

Revision	Description	Date
v1.4	Final version	08/18/2025
v1.3	Add verifications and other notes	08/01/2025
v1.2	Add one more items	07/29/2025
v1.1	Add report for the new tag	07/28/2025
v1.0	Draft report	07/25/2025

Executive Summary

The Sumer team (Sumer) has shared their smart contract source code on GitHub. We have listed the tag and the commit of the smart contracts to ensure that the audit results can be definitively linked to specific contract versions. The Tikkala research team collaborated with the Sumer team to address all potential findings and issues. The audit scope encompassed checking for vulnerabilities in smart contracts, including re-entry attacks, logic flaws, authentication bypasses, and DoS attacks, among others. Our researchers primarily focused on the changes except the Oracle related.

Disclaimer

Please note that security audit services cannot guarantee the discovery of all potential security issues within smart contracts. It is advisable to conduct repeated or incremental audits. Engaging multiple auditors for several audits is recommended. Product owners should maintain their own set of test cases and implement a regular code review process. Employing a threat intelligence system can aid in identifying or thwarting potential attacks, thereby reducing risk. Moreover, initiating a bug bounty program with the community can significantly enhance product security. Lastly, remember that security is complex! Even a robust smart contract does not ensure that your product is immune to all cybersecurity threats.
This audit focuses solely on identifying potential security vulnerabilities, such as code exploits and access control issues, in the smart contract or system. It does not evaluate business logic, economic models, or design feasibility. Specifically, the auditor does not assess the mathematical soundness, viability, or strength of rewarding mechanisms, tokenomics, or incentives, as these require expertise in economics or game theory beyond this security audit..

Contracts overview

The repo and tags audited this time are listed below(in this scope):

Repo	Tags	Revision
sumer-project	refs/tags/tikkala-audit-ready	12b988ee7b6c367eadc96b96935d8e3d6461f146
	refs/tags/tikkala-audit-fix-1	f874c0ffcaec18237adfb83abb34e602138a2a9a
	refs/tags/tikkala-audit-fix-2	5e39d6444b63e4a56d6c47ddc80d152d605280df

The findings

Results

ID	Description	Severity	Product Impact	Status
Sumer-01	User may claim more Rewards	Critical	Critical	Fixed
Sumer-02	Better update state before calling external functions	Info	Info	Fixed
Sumer-03	In-consistance between mint amount and stake amount	Low	Low	Fixed
Sumer-04	sweepToken() not work as expected	Low	Low	Fixed
Sumer-05	Be aware of function return value changes	Info	Info	N/A
Sumer-06	_syncUnderlyingBalance() is not working for Infrared CToken	Low	Low	Fixed

Details

Sumer-01 [Critical] User may claim more Rewards

In the reward model, whenever the collateral (CToken) balance changes, the `_userPoints` must be updated accordingly. However, this update may not occur

during the external call to `borrowAndDepositBack()`. Specifically, the internal function `borrowAndDepositBackInternal()` directly invokes `mintFresh()`, which updates the CToken balance but fails to refresh `_userPoints`. This oversight could enable a scenario where a user borrows and then deposits a large amount of tokens, allowing them to claim more rewards than intended.

```
1426  function borrowAndDepositBack(address borrower⬆, uint256 borrowAmount⬆) external nonReentrant {
1427    // only allowed to be called from su token
1428    if (CToken(msg.sender).isCToken()) {
1429      revert NotSuToken();
1430    }
1431    // only cToken has this function
1432    if (!isCToken()) {
1433      revert NotCToken();
1434    }
1435    if (!IComptroller(comptroller).isListed(msg.sender)) {
1436      revert MarketNotListed();
1437    }
1438    if (!IComptroller(comptroller).isListed(address(this))) {
1439      revert MarketNotListed();
1440    }
1441    borrowAndDepositBackInternal(payable(borrower⬆), borrowAmount⬆);
1442  }
1443
1444  /**
1445   * @notice Sender borrows assets from the protocol and deposit all of them back to the protocol
1446   * @param borrowAmount The amount of the underlying asset to borrow and deposit
1447   */
      ftrace | funcSig
1448  function borrowAndDepositBackInternal(address payable borrower⬆, uint256 borrowAmount⬆) internal {
1449    accrueInterest();
1450    borrowFresh(borrower⬆, borrowAmount⬆, false);
1451    mintFresh(borrower⬆, borrowAmount⬆, false);
1452  }
```

Suggestion: Update `_userPoints` accordingly.

Update: Fixed

Sumer-03 [Low] In-consistance between mint amount and stake amount

The contract `CInfraredVault` implements `doTransferIn()` and it uses the current balance as the staking amount. However, in some cases, the transfer in `finalAmount` may not match the staked amount, leading to an in-consistency issue.

```
function doTransferIn(
  address from⬆,
  uint256 amount⬆
) internal virtual override whenNotInVaultContext returns (uint256) {
  uint256 finalAmount = super.doTransferIn(from⬆, amount⬆);

  ERC20 underlyingToken = ERC20(underlying);
  uint256 balance = underlyingToken.balanceOf(address(this));
  underlyingToken.safeApprove(address(infraredVault), balance);
  infraredVault.stake(balance);

  return finalAmount;
}
```

Suggestion: Use the `finalAmount`

Update: Fixed

Sumer-04 [Low] sweepToken() not work as expected

The `sweepToken()` function in the `CErc20` contract previously verified the underlying token balance before and after a transfer to ensure the operator performed it correctly. However, the updated `sweepToken()` function removes this check, making it successful only when transferring zero tokens.

```
function sweepToken(ERC20 token⬆) external virtual override onlyAdmin {
  if (address(token⬆) == underlying) {
    revert CantSweepUnderlying();
  }
  uint256 underlyingBalanceBefore = token⬆.balanceOf(address(this));
  uint256 balance = token⬆.balanceOf(address(this));
  token⬆.safeTransfer(admin, balance);
  uint256 underlyingBalanceAfter = token⬆.balanceOf(address(this));
  if (underlyingBalanceBefore != underlyingBalanceAfter) {
    revert UnderlyingBalanceError();
  }
}
```

Suggestion: Still use the `underlying` to determine the balance.

Update: Fixed

Sumer-06 [Low] _syncUnderlyingBalance() is not working for Infrared CToken

In the function `_syncUnderlyingBalance()`, it provides a way to allow the admin force to sync the underlying balance, however, it is no longer working in staking based CToken.
This function should be disabled or it will cause a critical issue in the future.

```
1313  function _syncUnderlyingBalance() external virtual onlyAdmin {
1314    underlyingBalance = ICToken(underlying).balanceOf(address(this));
1315  }
```

Suggestion: Deprecated or disable this for Infrared CToken

Update: Fixed

Sumer-02 [Info] Better update state before calling external functions

By following the `CEI`(Checks-Effects-Interactions) security practice Checks-Effects-Interactions, it recommends to validate inputs, update state then call externally contract functions. In the function `_claimAllInternal()`, it is better to update `_userPoints` and `_totalPoints` earlier before calling token transfer.
The external functions which call `_claimAllInternal()` have re-entry guard, so this will be an informational only item.

```
188  function _claimAllInternal() internal {
189    _claimInfraredRewards();
190    address user = msg.sender;
191    uint256 pointsClaimed = _userPoints[user].amount;
192
193    if (_totalPoints < pointsClaimed) {
194      return;
195    }
196
197    address[] memory rewardTokens = getRewardTokens();
198    uint256 len = rewardTokens.length;
199    for (uint256 i; i < len; i++) {
200      ERC20 token = ERC20(rewardTokens[i]);
201      uint256 totalRewards = token.balanceOf(address(this)) - protocolRewards[rewardTokens[i]];
202      uint256 reward = (pointsClaimed * totalRewards) / _totalPoints;
203      if (reward > 0) {
204        token.safeTransfer(user, reward);
205        emit UserRewardPaid(user, address(token), reward);
206      }
207    }
208
209    _userPoints[user].amount -= 0;
210    _totalPoints = _totalPoints - pointsClaimed;
211  }
```

Suggestion:

Update: Fixed

Sumer-05 [Info] Be aware of function return value changes

Several functions in the contracts have had their return values removed or modified, which could create compatibility issues for contracts interacting with the Sumer project. For instance, the `liquidateCalculateSeizeTokens()` function's return value was changed from (`err`, `seizeTokens`, `seizeProfitTokens`) to (`seizeTokens`, `seizeProfitTokens`), eliminating the `err` value. This alteration may lead to unexpected behavior in dependent contracts or DApps that rely on the original return structure.

```
1013   function liquidateCalculateSeizeTokens(
1014     address cTokenCollateral,
1015     uint256 actualRepayAmount
1016 -  ) public view returns (uint256, uint256, uint256) {
1017 +  ) public view returns (uint256, uint256) {
1018     (bool repayListed, uint8 repayTokenGroupId, ) = IComptroller(comptroller).markets(address(this));
1019   ......
1020 @@ -1473,7 +1413,7 @@
1021     uint256 seizeTokens = mul_ScalarTruncate(ratio, actualRepayAmount);
1022     uint256 seizeProfitTokens = mul_ScalarTruncate(profitRatio, actualRepayAmount);
1023
1024 -   return (uint256(0), seizeTokens, seizeProfitTokens);
1025 +   return (seizeTokens, seizeProfitTokens);
1026   }
```

Suggestion: For the external/public function, it is better not to change the return value format if the return values are crucial.

Update: N/A

Summary

The Tikkala research team conducted both automated and manual audits on the Sumer smart contracts listed above. All identified issues were communicated to the Sumer team via a Telegram channel prior to this report. The audit uncovered 1 critical,2 low, and 2 informational impact issues. The Sumer team responded promptly, addressing all the issues.The Tikkala research team then verified and confirmed these fixes on GitHub.