AUDIT REPORT

Sumer.Money

September 2023

Introduction

A time-boxed security review of the **Sumer.Money** protocol was done by **ddimitrov22** and **chrisdior4**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Sumer.Money**

Sumer lending and borrowing market is a Compound-inspired protocol that enables:

- Deposit of Native Assets or their equivalent
- Minting of SuTokens(SuUSD, SuETH, etc.) against the deposited native assets
- Borrowing native assets against deposited native assets
- Borrowing native assets against deposited SuTokens
- Repayment of borrowed assets
- Liquidation

The Sumer Protocol will mainly accept on-chain stablecoins and blue-chip assets (ETH, BTC, BNB, etc.) as collateral in the initial deployment of the protocol.

The protocol has adapted a variable interest rate model where the rates are largely determined by the supply and demand of assets, represented by its Utilization rate. The computation of interest rates is separated into two stages, the Standard model and the Jump (Kink) model, to further incentivize/disincentivize depositing and borrowing activity.

More docs

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 02d76881ca5617b86881849ef833cd233e436908**

Scope

All of the contracts were in scope of the audit except the below:

- `ERC20/..`
- `proxy/..`

The following number of issues were found, categorized by their severity:

- Critical & High: 1 issues
- Medium: 7 issues
- Low: 3 issues
- Informational: 10 issues

Findings Summary

ID	Title	Severity
[H-01]	Incorrect `blocksPerYear` constant leads to wrong calculations	High
[M-01]	Possible incorrect utilization rate	Medium
[M-02]	Borrow rate calculation can cause `CToken.accrueInterest()` to revert	Medium
[M-03]	Insufficient input validation	Medium
[M-04]	Use `call()` instead of `transfer()` when sending ETH	Medium
[M-05]	Assumption that all tokens have 18 decimals will lead to wrong calculations	Medium
[M-06]	Use of `transferFrom` is dangerous for non-compliant ERC20 tokens	Medium
[M-07]	A malicious/compromised `owner` of `TokenManager` can rug rewards	Medium
[L-01]	Check if array arguments have the same length	Low
[L-02]	Single-step ownership transfer pattern is dangerous	Low
[L-03]	Unbounded for-loop can run out of gas	Low
[I-01]	Use `address.code.size` attribute	Informational
[I-02]	Wrong comments	Informational
[I-03]	Unused modifier	Informational

ID	Title	Severity
[I-04]	NatSpec docs are incomplete	Informational
[I-05]	Repeating code can be made into a modifier	Informational
[I-06]	Open TODO in the code	Informational
[I-07]	Prefer Solidity Custom Errors over `require` statements with strings	Informational
[I-08]	Missing event emissions in state changing methods	Informational
[I-09]	Using `SafeMath` when compiler is ^0.8.0	Informational
[I-10]	Use newer Solidity version with a stable pragma statement	Informational

Detailed Findings

[H-01] Incorrect `blocksPerYear` constant leads to wrong calculations

Severity

Impact: High, because critical functions of the protocol are affected

Likelihood: Medium, because it will happen only for certain blockchains

Description

Incorrect `blocksPerYear` constant in `JumpRateModel` leads to wrong calculations. As discussed with the dev team, and as can be seen from the docs, the protocol will be deployed on L2 chains (Meter,Base) as well as on the ETH Mainnet. The `JumpRateModel` is forked from Compound which was designed to be deployed on the ETH Mainnet. The problem is that the `blocksPerYear` constant which is used to calculate the interest rate on a per-block basis is set to 2102400:

```
uint256 public constant blocksPerYear = 2102400;
```

The number assumes that the block time is 15 seconds which is wrong for the L2 chains. The average block time on Meter is `1.93 sec` while on BASE is `2 sec` hence the `blocksPerYear` number should be much higher. Because of that the `baseRatePerBlock` and `multiplierPerBlock` will be affected and will be much higher. This will lead to inflating `getBorrowRate` which directly affects other critical functions of the protocol.

Recommendations

Depending on which chain the project is deployed, pass the value for `blocksPerYear` in the constructor as it is done for the other interest rate models.

Client

Acknowledged - The client decided to not use this rate model

[M-01] Possible incorrect utilization rate

Severity

Impact: High, as this will lead to getting both supply rate and borrow rate calculations wrong

Likelihood: Low, because the reserves grow automatically over time

Description

In `BaseJumpRateModelV2.sol:utilizationRate()`, `cash`, `borrows` and `reserves` values are used to calculate the utilization rate. If `borrows` value is 0, then function will return 0. But in this function the scenario where the value of `reserves` exceeds cash is not handled. The system does not guarantee that `reserves` never exceeds `cash`. The `reserves` grow automatically over time, so it might be difficult to avoid this entirely.

```
function utilizationRate(
        uint256 cash,
        uint256 borrows,
        uint256 reserves
    ) public pure returns (uint256) {
        // Utilization rate is 0 when there are no borrows
        if (borrows == 0) {
            return 0;
        }

        return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
    }
```

If `reserves` > `cash` (and `borrows` + `cash` - `reserves` > 0), the formula for utilizationRate above gives a utilization rate above 1.

Recommendations

Make the utilization rate computation return 1e18 (which is the maximum utilization rate) if `reserves` > `cash`.

Client

Fixed

[M-02] Borrow rate calculation can cause `CToken.accrueInterest()` to revert

Severity

Impact: High, as this will cause a major DoS to the functions in CToken that are calling `accrueInterest()` internally

Likelihood: Low, because there is a chance that `borrowRateMantissa` will be bigger than the hardcoded value

Description

CToken hard-codes the maximum borrow rate `BORROW_RATE_MAX_MANTISSA`. Then we have `accrueInterest()` which reverts if the dynamically calculated rate `borrowRateMantissa` is greater or equal than the hard-coded one.

Here is the check from `accrueInterest()`:

```
require(borrowRateMantissa <= BORROW_RATE_MAX_MANTISSA, 'borrow rate is
absurdly high');
```

The actual calculation is dynamic and takes no notice of the hard-coded cap, so it is very possible that this state will manifest, causing a major DoS due to most CToken functions calling `accrueInterest()` and `accrueInterest()` reverting.

Recommendations

Change `CToken.accrueInterest()` to not revert in this case, but simply to set `borrowRateMantissa = borrowRateMaxMantissa` if the dynamically calculated value would be greater than the hard-coded max. This would:

- Allow execution to continue operating with the system-allowed maximum borrow rate, allowing all functionality that depends upon `accrueInterest()` to continue as normal.
- Allow `borrowRateMantissa` to be naturally set to the dynamically calculated rate as soon as that rate becomes less than the hard-coded max.

Client

Fixed

[M-03] Insufficient input validation

Severity

Impact: High, because some of these functions are setting important values which can be problematic

Likelihood: Low, as it requires a malicious/compromised owner account or an owner input error

Description

Throughout the codebase there are couple of places where importan input validation is missing. The inputs are not constrained at all or only partly.

- The `newCloseFactorMantissa` param in `Comptroller:_setCloseFactor()`.
- The `supplySpeed` and `borrowSpeed` params in `CompLogic:_setCompSpeedInternal()`.
- The `_rewardsDuration` param in `StakingRewardsMultiGauge:setRewardsDuration()` is only partly validated but it is missing an upper constrain.
- The `new_rate` param in `StakingRewardsMultiGauge:setRewardRate()`.

Recommendations

You can create a check where `exampleParam` should be less than x or greater than y, otherwise revert the transaction.

Client

Fixed

[M-04] Use `call()` instead of `transfer()` when sending ETH

Severity

Impact: Medium, because if the recipient is a smart contract or multisig the trx will fail

Likelihood: Medium, because there is a big chance that the recipient will be a smart contract or a specific multisig wallet that requires more than 2300 gas

Description

The `repayBorrowBehalf()` is using the `transfer` method to send ETH to the caller of the function. This address is possible to be a smart contract that has a `receive` or `fallback` function that takes up more than the 2300 gas which is the limit of `transfer`. Examples are some smart contract wallets or multi-sig wallets, so usage of `transfer` is discouraged.

More

Recommendations

Use a `call` with value instead of `transfer`.

Client

Fixed

[M-05] Assumption that all tokens have 18 decimals will lead to wrong calculations

Severity

Impact: Medium, because the loss for either the protocol or the user will not be significant

Likelihood: Medium, because only tokens with less/more than 18 decimals will be affected

Description

There are functions related to calculating rewards that assumes all tokens involved have 18 decimals. Such functions are rewardsPerToken and earned inside `CommunalFarm`:

```
function rewardsPerToken() public view returns (uint256[] memory
newRewardsPerTokenStored) {
...
} else {
      newRewardsPerTokenStored = new uint256[](rewardTokens.length);
      for (uint256 i = 0; i < rewardsPerTokenStored.length; i++) {
        newRewardsPerTokenStored[i] = rewardsPerTokenStored[i].add(

lastTimeRewardApplicable().sub(lastUpdateTime).mul(rewardRates[i]).mul(1e1
8).div(_total_combined_weight) //@audit mul(1e18) assumes all tokens have
18 decimals
...
}
```

```
function earned(address account) public view returns (uint256[] memory
new_earned) {
 ...
for (uint256 i = 0; i < rewardTokens.length; i++) {
        new_earned[i] = (_combined_weights[account])
          .mul(reward_arr[i].sub(userRewardsPerTokenPaid[account][i]))
          .div(1e18)
          .add(rewards[account][i]);
      }
    }
 }
```

This is problematic because the protocol intends to use `USDC`` and` USDT``tokens but they both have 6 decimals. This leads to wrong reward calculations and effectively loss of funds for all pools that will be using tokens with different decimals than`18`.

Recommendations

Add support for different number of decimals than `18` by dynamically checking `decimals()` for the tokens that are part of the rewards calculations.

Client

Acknowledged - only LP tokens will be staked, which have 18 decimals

[M-06] Use of `transferFrom` is dangerous for non-compliant ERC 20 tokens

Severity

Impact: Medium, because token transfers can fail silently

Likelihood: Medium, because there are such ERC 20 tokens which the protocol intends to implement

Description

The ERC 20 `transfer` and `transferFrom` functions return a boolean indicating success which needs to be checked. However, some tokens do not revert if the transfer failed. Such tokens are `USDC` and `USDT` which are non-compliant to the ERC 20 standard. This could lead to stuck funds in the contract. An example of such implementation is inside `CommunalFarm`:

```
function _getReward(
    address rewardee,
    address destination_address
  ) internal updateRewardAndBalance(rewardee, true) returns (uint256[]
memory rewards_before) {
    ...
     IERC20(rewardTokens[i]).transfer(destination_address,
rewards_before[i]); //@audit use safeTransfer
     emit RewardPaid(rewardee, rewards_before[i], rewardTokens[i],
destination_address);
    ...
```

Recommendations

Use `safeTransfer` of SafeERC20.

Client

Fixed

[M-07] A malicious/compromised `owner` or `TokenManager` can rug rewards

Impact: High, as it will result in direct theft of funds from users

Likelihood: Low, as malicious/compromised owner or manager are required

Description

The `recoverERC20` functions are intended to allow the `owner` and the `TokenManager` to withdraw any mistakenly tokens sent to the system:

```
unction recoverERC20(address tokenAddress, uint256 tokenAmount) external
onlyTknMgrs(tokenAddress) {
    // Cannot rug the staking / LP tokens
    require(tokenAddress != address(stakingToken), 'Cannot rug staking /
LP tokens');

    // Check if the desired token is a reward token
    bool isRewardToken = false;
    for (uint256 i = 0; i < rewardTokens.length; i++) {
      if (rewardTokens[i] == tokenAddress) {
        isRewardToken = true;
        break;
      }
    }

    // Only the reward managers can take back their reward tokens
    if (isRewardToken && rewardManagers[tokenAddress] == msg.sender) {
      IERC20(tokenAddress).transfer(msg.sender, tokenAmount);
      emit Recovered(msg.sender, tokenAddress, tokenAmount);
      return;
    }
    // Other tokens, like airdrops or accidental deposits, can be
withdrawn by the owner
    else if (!isRewardToken && (msg.sender == owner())) {
      IERC20(tokenAddress).transfer(msg.sender, tokenAmount);
      emit Recovered(msg.sender, tokenAddress, tokenAmount);
      return;
    }
    // If none of the above conditions are true
    else {
      revert('No valid tokens to recover');
    }
  }
```

However, in the instance above which is inside the `CommunalFarm` contract, the `TokenManager` can withdraw any amount of the reward tokens without any restrictions. This could lead to a direct theft of all of the reward tokens.

Client

Acknowledged

Recommendations

Consider allowing the privileged roles to be able to withdraw only tokens that are not reward tokens or the `stakingToken`.

[L-01] Check if array arguments have the same length

In the constructor of `StakingRewardsMultiGauge`, we have couple of array-type arguments. Two of them (`_rewardTokens`, `_rewardManagers`) are later included in a for loop inside the constructor. Validate that the arguments have the same length so you do not get unexpected errors if they don't.

```
rewardManagers[_rewardTokens[i]] = _rewardManagers[i];
```

Client

Fixed

[L-02] Single-step ownership transfer pattern is dangerous

We can observe the inheritance of `Ownable` in couple of the contracts in the codebase, for example `CommunalFarm.sol`. Inheriting from OpenZeppelin's `Ownable` contract means you are using a single-step ownership transfer pattern. If an admin provides an incorrect address for the new owner this will result in none of the `onlyOwner` marked methods being callable again. The better way to do this is to use a two-step ownership transfer approach, where the new owner should first claim its new rights before they are transferred. Use OpenZeppelin's `Ownable2Step` instead of `Ownable`.

Client

Fixed

[L-03] Unbounded for-loop can run out of gas

There are several places across the codebase where a looping over an array of tokens is happening. Example of that is the `rewardsToken` array:

```
for (uint256 i = 0; i < rewardTokens.length; i++) { //@audit possible OOG
error if the array grows too big
...
}
```

If the array grows too big it is possible the transaction to exceed the block gas limit and revert resulting in a DoS. Consider to cap the `rewardsToken` array to 20 for example.

Client

Acknowledged

[I-01] Use `address.code.size` attribute

Solidity version 0.8.13 introduced an address.code.size attribute, which is equivalent to calling the extcodesize opcode in inline assembly. It can be used to simplify the `isContract` function in `Comptroller.sol`:

```
function isContract(address account) internal view returns (bool) {
    // This method relies on extcodesize, which returns 0 for contracts in
    // construction, since the code is only stored at the end of the
    // constructor execution.

    uint256 size;
    // solhint-disable-next-line no-inline-assembly
    assembly {
      size := extcodesize(account)
    }
    return size > 0;
  }
```

Change it to:

```
 function isContract(address account) internal view returns (bool) {
        return address.code.size > 0;
    }
```

Client

Fixed

[I-02] Wrong comments

1. First one is in `CToken.sol`:

```
/* Fail if repayAmount = -1 */
if (repayAmount == ~uint256(0)) {
```

Here we can see that the comment says that the check should fail if `repayAmount` is equal to `-1`. But `~uint256(0)` is expressing the maximum value of uint256. Either change the comment or the code.

2. `StakingRewardsMultiGauge.sol`:

```
// FRAX
address public constant usd_address =
0x0d893C092f7aE9D97c13307f2D66CFB59430b4Cb;
```

This comment is supposedly pointing that the below declared address is of FRAX but is actually USD. Change this.

3. `StakingRewardsMultiGauge.sol`:

```
// Get the amount of FRAX 'inside' of the lp tokens
   uint256 usd_per_lp_token;
```

Almost the same as the above one.

Client

Partially fixed

[I-03] Unused modifier

In `CommunalFarm.sol` and `StakingRewardsMultiGauge.sol` there is a modifier `notStakingPaused` that is not used. Either make use of it or remove it.

Client

Fixed

[I-04] NatSpec docs are incomplete

Some external methods are missing NatSpec documentation which is essential for better understanding of the code by developers and auditors and is strongly recommended. Please refer to the NatSpec format and follow the guidelines outlined there.

An example is `setComptroller()` in `AccountLiquidity.sol`.

Client

Acknowledged

[I-05] Repeating code can be made into a modifier

In `CompLogic.sol` we have the following require statement which is used 6 times in different places throughout the contract:

```
require(msg.sender == address(comptroller), 'forbidden!');
```

To avoid the repeating they can be made into a modifier which will make the code look more neat and organised. Also it will improve the readability.

Client

Fixed

[I-06] Open TODO in the code

There is open `TODO`s in `CompLogic.sol` and `FeedPriceOracle.sol` which shows that code is not production-ready. Resolve it or remove it.

Client

Fixed

[I-07] Prefer Solidity Custom Errors over `require` statements with strings

You are using mostly require/revert now. Using Solidity Custom Errors has the benefits of less gas spent in reverted transactions, better interoperability of the protocol as clients of it can catch the errors easily on-chain, as well as you can give descriptive names of the errors without having a bigger bytecode or transaction gas spending, which will result in a better UX as well. Consider replacing the `require` statements with custom errors.

Client

Acknowledged

[I-08] Missing event emissions in state changing methods

It's a best practice to emit events on every state changing method for off-chain monitoring. Some examples will be `setComptroller()` in `AccountLiquidity.sol` and `setTimelock()`, `setCurator()`, `setGaugeController()` in `FraxGaugeFXSRewardsDis.sol`.

Client

Acknowledged

[I-09] Using `SafeMath` when compiler is ^0.8.0

There is no need to use `SafeMath` when compiler is ^0.8.0 because it has built-in under/overflow checks. Also you are both using methods from `SafeMath` and the normal arithmetic operators such as `*`, `/`, etc. Use them instead of `SafeMath` functions.

Client

Acknowledged

[I-10] Use newer Solidity version with a stable pragma statement

Some contracts are using a floating pragma statement which is discouraged as code can compile to different bytecodes with different compiler versions. Use a stable pragma statement to get a deterministic bytecode. Consider using a stable 0.8.19 version to make sure it is up to date in all of the contracts. Example is: `FraxGaugeFXSRewardsDistributor.sol`:

```
pragma solidity >=0.8.19;
```

Client

Acknowledged