SENTIMENT PROTOCOL SECURITY ASSESSMENT

July 27, 2022

Prepared For:

0xSnarks

Prepared By:

Chris Masden

Alexis Williams

Ian Bridges

Changelog:

July 1, 2022	*Initial report delivered*
July 27, 2022	*Final report delivered*

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This report contains the results of Arbitrary Execution's security assessment of Sentiment's protocol smart contracts. Sentiment is a permissionless, under-collateralized, on-chain credit system that allows users to post assets as collateral in exchange for loans. The protocol uses smart contract `Accounts` to hold collateral and loans, `Controller` contracts to interact with external protocols, and `Oracles` to receive pricing data for assets. Borrowers in the Sentiment ecosystem create an account which holds collateral and loaned assets and actions are performed on that account by the Account Manager. This allows a borrower the freedom to determine how assets are used without having custody. Additionally, having collateral and loaned assets in a standalone `Account` ensures borrowers are unable to withdraw loaned assets without having sufficient collateral left in an `Account`. This effectively 'sandboxes' the Sentiment Protocol. The delegated ownership of an account is a core function of the protocol as this ensures that essential actions can be performed on an account such as a liquidation. Lenders are expected to supply liquidity to the aforementioned borrowers in order to receive interest bearing `LTokens` which can be burned to redeem the principal and accrued interest.

Three Arbitrary Execution (AE) engineers conducted this review over a 3-week period, from June 6, 2022 to June 24, 2022. The audited commit for the protocol contracts was `4c97a4aa8e6d69507a53ee8cd5ffaa3f4af4c59f` in the `main` branch of the `sentimentxyz/protocol` repository. This repository was private at the time of the engagement, so hyperlinks may not work for readers without access. The solidity files in the `src/core`, `src/interface`, `src/utils`, `src/proxy`, and `src/tokens` directories were in scope for this audit. The complete list of files is located in Appendix B.

The team performed a detailed, manual review of the codebase with a focus on the core protocol contracts. These contracts provide the necessary logic to run the Sentiment Protocol. The protocol contracts also interface with the Controller and Oracle contracts which were audited in a previous effort. In addition to manual review, the team used Slither for automated static analysis.

FIX REVIEW UPDATE

FIX REVIEW PROCESS

After receiving fixes for the findings shared with Sentiment, the AE team performed a review of each fix. Each pull request was scrutinized to ensure that the core issue was addressed, and that no regressions were introduced with the fix. A summary of each fix review can be found in the *update* section for a finding. For findings that the Sentiment team chose not to address, the team's rationale is included in the update.

The Sentiment team provided fixes for 33 findings, 2 partially fixed finding, and has acknowledged the remaining 3 issues. All critical, medium, and low severity findings were fixed by the Sentiment team with the exception of `L06` for which they haven an open pull request and are actively addressing. The [`L07`] issue was addressed by implementing a timelock on a controlling multisig wallet. The multisig wallet and its timelock were considered out of scope for this audit. The full breakdown of fixes can be found in the Fixes Summary section.

OBSERVATIONS

Three critical severity findings were found during the security assessment. The first allows a malicious user to create a scenario where they are able to take control of a user's account and extract all of their tokens. This is done by using reentrancy to close an account twice and then waiting for a new user to open a new account. Once this occurs, the malicious user can open a new account and they will be made the owner of the account the new user just made. Another critical finding was that the liquidation logic in the `AccountManager` contract does not contain logic to allow a maintainer to repay the outstanding borrows on an account that is eligible for liquidation. This could lead to accounts that are protected from liquidation when the account does not have enough tokens to pay off its debt. The third critical finding is that ERC-20 balances are calculated incorrectly. This leads to ERC-20 tokens that have more than 18 decimals being greatly overvalued, and ERC-20 tokens that have less than 18 decimals being greatly undervalued. The core reason this vulnerability exists is that when calculating the balance of an account (and when calculating the value of all current borrows) token decimals are not checked and are assumed to be in 18 decimals. This can lead to a malicious user borrowing far more than they should be able to and can also lead to user positions being liquidated unfairly.

The core logic in the Sentiment protocol that allows under-collateralized loans to work is the concept of delegated ownership. The end user using a protocol does not hold the assets, instead they are held by an `Account` contract. The `Account` contract is operated on by the `AccountManager`, and it is through the `AccountManager` that account owners can perform operations such as `deposit`, `borrow`, `repay`, and `exec`. The `exec` function in the `AccountManager` is the function that allows users to interact with a limited number of external contracts, such as other DeFi protocols like Compound or Aave, using their account that holds both collateral and borrowed assets. The Sentiment Protocol also uses a series of `Controller` contracts (that were not in scope for this audit) to control what external contracts and functions a user is allowed to interact with. Additionally, the `AccountManager` uses the `RiskEngine` contract to ensure that an account stays within a specified health factor before and after actions involving assets are performed on an account. This controlled ecosystem and enforcement of maintaining healthy positions by an account helps protect the Sentiment Protocol and its lenders against the inherent risk of under-collateralized lending.

These abstractions, however, make interacting with external protocols more difficult as the users are ultimately in charge of crafting the calldata necessary to interact with external protocols. Additionally, the lack of error and return data propagation in `exec` means users would need to debug internal trace data in the event that an external protocol interaction fails as the high-level transaction will always succeed. As such, the Sentiment Protocol should be considered an advanced protocol suitable for users who have a strong understanding of how transactions work and should be able to debug internal trace data if necessary.

Also, external dependencies were used throughout the codebase, but these dependencies were not installed with any package manager. This makes it difficult to determine which version of the code is being used. It also makes updating dependencies in the event of a bug fix difficult as there would be no notification or easy way to upgrade; a manual copy and paste would need to be performed, which can also lead to unintentional bugs. One of the dependencies used is the `solmate` repository from Rari-Captial. Rari-Captial specifically calls out the following about `solmate`:

> *This is experimental software and is provided on an "as is" and "as available" basis.*
>
> *While each major release has been audited, these contracts are not designed with user safety in mind:*
>
> *There are implicit invariants these contracts expect to hold. You can easily shoot yourself in the foot if you're not careful. You should thoroughly read each contract you plan to use top to bottom. We do not give any warranties and will not be liable for any loss incurred through any use of this codebase.*

These are gas optimized contracts that lack many safety checks that OpenZeppelin libraries provide. The Sentiment team should carefully consider the trade offs of choosing either method. Another benefit of using a package manager to track external dependencies is that it reduces the code base size. For example, `IERC20.sol` is an interface used to interact with ERC-20 tokens and the file is located at `src/interface/tokens/IERC20.sol` which isn't necessary as the OpenZeppelin `IERC20.sol` file could be used and `src/interface/tokens/IERC20.sol` could be removed.

There is extensive unit test coverage for the underlying protocol contracts. However, when looking at the tests and the way they are run a few problems were observed. The first is that the test coverage on edge cases could be improved. Many tests provide expected input to the functions and do not try to cover abnormal behavior. An example of this could be the critical finding where a malicious user can perform a hostile takeover on an account. Another potential area of concern is how some of the tests are written. Some of the tests can pass for the wrong reason. One example of this is `testLiquidationEth` in `LiquidationFlow.t.sol`. The assertions in the test pass because the balance of the account to be liquidated holds enough of the borrowed tokens to successfully repay the borrow irrespective to the amount of `ether` the maintainer (i.e. the caller of the liquidate function) currently has. This can be displayed by setting the maintainer `ether` balance to 0 (`cheats.deal(maintainer, 0);`) and then performing the liquidation. The liquidation will incorrectly succeed even though the liquidator did not provide any funds for liquidation.

VULNERABILITY STATISTICS

Severity	Count
Critical	3
High	0
Medium	3
Low	8
Note	24

FIXES SUMMARY

Finding	Severity	Status
C01	Critical	Fixed in pull request #216
C02	Critical	Fixed in pull request #193
C03	Critical	Fixed in pull request #192
M01	Medium	Fixed in pull request #194
M02	Medium	Fixed in pull request #189
M03	Medium	Fixed in pull request #195
L01	Low	Fixed in pull request #192
L02	Low	Fixed in pull request #196
L03	Low	Fixed in pull request #193
L04	Low	Fixed in pull request #197
L05	Low	Fixed in pull request #199
L06	Low	Acknowledged
L07	Low	Fixed
L08	Low	Fixed in pull request #198
N01	Note	Acknowledged
N02	Note	Fixed in pull request #200
N03	Note	Acknowledged
N04	Note	Fixed in pull request #201
N05	Note	Fixed in pull request #202
N06	Note	Fixed in pull request #204
N07	Note	Fixed in pull request #210
N08	Note	Partially fixed in pull request #203
N09	Note	Fixed in pull request #211
N10	Note	Fixed in pull request #203
N11	Note	Fixed in pull request #213
N12	Note	Fixed in pull request #220 and pull request #213

N13	Note	Fixed in pull request #209
N14	Note	Partially fixed in pull request #206
N15	Note	Fixed in pull request #208
N16	Note	Fixed in pull request #200
N17	Note	Fixed in pull request #214
N18	Note	Fixed in pull request #207
N19	Note	Fixed in pull request #205
N20	Note	Fixed in pull request #205
N21	Note	Fixed in pull request #205
N22	Note	Fixed in pull request #189
N23	Note	Fixed in pull request #219
N24	Note	Fixed in pull request #212

FINDINGS

CRITICAL SEVERITY

[C01] INCORRECT ERC-20 TOKEN VALUATION

The `_getBalance` function and `_getBorrows` function in `RiskEngine.sol` are the underlying `internal` implementation functions that correspond to their `external` counterparts `getBalance` and `getBorrows`, respectively. `_getBalance` iterates over an account's assets and sums the total value in `wei` of each token asset before finally adding the account's `ether` balance:

```
function _getBalance(address account) internal view returns (uint) {
    address[] memory assets = IAccount(account).getAssets();
    uint assetsLen = assets.length;
    uint totalBalance;
    for(uint i; i < assetsLen; ++i) {
        totalBalance += _valueInWei(
            assets[i],
            IERC20(assets[i]).balanceOf(account)
        );
    }
    return totalBalance + account.balance;
}
```

`_getBorrows` similarly iterates over each token an account has borrowed and sums the total borrow value (i.e. borrow balance plus interest) in `wei`:

```
function _getBorrows(address account) internal view returns (uint) {
    if (IAccount(account).hasNoDebt()) return 0;
    address[] memory borrows = IAccount(account).getBorrows();
    uint borrowsLen = borrows.length;
    uint totalBorrows;
    for(uint i; i < borrowsLen; ++i) {
        address LTokenAddr = registry.LTokenFor(borrows[i]);
        totalBorrows += _valueInWei(
            borrows[i],
            ILToken(LTokenAddr).getBorrowBalance(account)
        );
    }
    return totalBorrows;
}
```

Both functions use the `_valueInWei` function which calls out to Sentiment's `OracleFacade` contract (which was not in scope for this audit) and returns the price of the passed-in token in `wei`. This price is then multiplied by the passed-in token amount (which is labelled as `value` in the `_valueInWei` function):

```
function _valueInWei(address token, uint value)
    internal
    view
```

```
    returns (uint)
{
    return oracle.getPrice(token).mul(value);
}
```

However, both `_getBalance` and `_getBorrows` neglect to scale token balances to 18 decimals. This means tokens whose decimals are less than 18 will be greatly undervalued compared to tokens whose decimals are greater than or equal to 18. Additionally, tokens whose decimals are greater than 18 will be greatly overvalued compared to tokens whose decimals are less than or equal to 18.

For example, say a user has the assets `tokenA` and `tokenB` in their account, where the user has 1 of each token. Additionally, `tokenA` is worth 1 `ether` and has 8 decimals, and `tokenB` is also worth 1 `ether` but has 18 decimals. When `getBalance` in `RiskEngine.sol` is called to determine the value of the account's assets, the underlying `_getBalance` function will be called which will iterate over each token and sum the values. However, when `_valueInWei` is called to determine the value of each token, the token balance for `tokenA` will be `1 * 1e8` whereas the token balance for `tokenB` will be `1 * 1e18`. Subsequently, the number returned from `_valueInWei` for `tokenA` will be `(oracle price in wei * token balance) = (1 * 1e18) * (1 * 1e8) = 1 * 1e26` and the number returned from `_valueInWei` for `tokenB` will be `(1 * 1e18) * (1 * 1e18) = 1 * 1e36`. This means that even though a user's account has the same number of `tokenA` and `tokenB`, and both tokens are worth the same amount of `ether`, `tokenA` will be undervalued compared to `tokenB` because its decimals were not scaled to 18.

Consequently, the collateral a user provides may be undervalued or overvalued relative to the tokens a user wants to borrow. In the case where a user provides collateral tokens with less than 18 decimals, the token would be undervalued and therefore the user would only be able to borrow less than the expected amount. Additionally, if the value of the assets a user has in their account is undervalued relative to the borrow value of the account, the account may be unjustly liquidated where the maintainer who initiated the liquidation would receive the user's assets at a greatly reduced price. In the case where a user provides collateral tokens with greater than 18 decimals, the token would be overvalued and therefore a user would be able to borrow more than the expected amount.

RECOMMENDATION

Consider scaling the token balance for each asset or borrowed token to 18 decimals, and passing that number to `_valueInWei`.

UPDATE

Fixed in pull request [#216](#) (commit hash `bca40676b067f7e24f054f93961bee6a565dcdf3`), as recommended.

[C02] MAINTAINERS MAY NOT BE ABLE TO PERFORM LIQUIDATIONS

In order to minimize the inherent risk of undercollateralized lending, the Sentiment Protocol allows liquidations to be performed on accounts that fall below the `balanceToBorrowThreshold` of `12 * 1e17`, which is defined in the `RiskEngine` [contract](). The value checked against the `balanceToBorrowThreshold` for an account is the ratio of an account's balance relative to its borrows. This ratio is calculated by taking the total value of the assets an account possesses, to include the assets borrowed, and dividing that by the total value of an account's borrows plus interest. When the calculated ratio for an account falls below the `balanceToBorrowThreshold` aka `12 * 1e17`, a liquidation can then occur. According to Sentiment's [documentation]():

> *Liquidations will be akin to an OTC swap, whereby the maintainer pays the loan(s) on the Account's behalf, and receives Margin Account's portfolio at a discount.*

The `liquidate` [function]() is the function that gets called when an external address wants to liquidate an unhealthy account. The `liquidate` function uses the internal the internal `_liquidate` [function](), which iterates over the addresses of the account's borrowed tokens and calls `_repay` to repay each borrow. The internal `_repay` [function]() transfers the borrow balance plus interest on the borrowed asset from the account to the `LToken` vault contract, which is where the initial borrow was loaned from.

This is where the core problem lies: There is no logic in `_repay` to transfer tokens from the caller address, aka maintainer, that initiated the liquidation of the account to the account itself in order to pay off the borrows. As a result, in the situation where an account does not have enough borrowed tokens to repay their debts, a maintainer will be unable to liquidate the account as the `liquidate` call will revert due to insufficient funds.

The following scenario encapsulates how an account could get into a state where the `liquidate` function would revert:

1. A borrower opens a new account and deposits 1 `tokenA` as collateral. `tokenA` is currently worth 1 `ether` and has 18 decimals.
2. The borrower then borrows 1 `tokenB`, where `tokenB` is also currently worth 1 `ether` and has 18 decimals.
3. The borrower calls `exec` from `AccountManager.sol` to initiate a swap of 1 `tokenB` for 1 `tokenA`. The borrower's account now has 2 `tokenA` and 0 `tokenB`.
4. As time progresses, the value of `tokenA` decreases to 0.25 `ether`, but the value of `tokenB` stays the same at 1 `ether`. The ratio of the account's balance value to borrows value is now `((0.25 * 1e18) * (2 * 1e18)) / ((1 * 1e18) * (1 * 1e18)) = 0.5 * 1e18`. The value `0.5 * 1e18` is less than the `balanceToBorrowThreshold` of `12 * 1e17`, so the account is now liquidatable. The interest calculation on total borrows was abstracted away for simplicity.
5. A maintainer notices that an account is liquidatable, and calls `liquidate` on the borrower's account.
6. The underlying `_repay` function calculates the borrow balance plus interest that the account needs to repay and then attempts to transfer that amount from the liquidatable account to the `LToken` vault. The underlying `transfer` call reverts because the liquidatable account has 0 `tokenB` to transfer. There is no logic in `liquidate` or any other of the underlying functions to transfer `tokenB` that the maintainer holds to the account so it can pay off the account's debts.
7. The `liquidate` function call fails due to the revert in `transfer` and the maintainer is unable to liquidate the account.

RECOMMENDATION

Consider adding logic that transfers tokens from the liquidator address to the liquidatable account when the `liquidate` function is called.

UPDATE

Fixed in pull request #193 (commit hash `7c5a5182ee0c002ad419275b3d2d2229a01ba227`), as recommended. The ERC-20 tokens needed to liquidate an account now come from the liquidator. It should be noted that in order for a liquidation to succeed, a prior approval from the liquidator to the `AccountManager` contract must exist on the ERC-20 token that is being repaid.

[C03] POSSIBLE ACCOUNT TAKEOVER BY A MALICIOUS USER

Account owners are able to close their account by calling the `closeAccount` function in `AccountManager.sol`:

```
function closeAccount(address _account) public onlyOwner(_account) {
    IAccount account = IAccount(_account);
    if (account.activationBlock() == block.number)
        revert Errors.AccountDeactivationFailure();
    if (!account.hasNoDebt()) revert Errors.OutstandingDebt();
    account.sweepTo(msg.sender);
    registry.closeAccount(_account);
    inactiveAccounts.push(_account);
    emit AccountClosed(_account, msg.sender);
}
```

`closeAccount` first goes through a series of checks to make sure that:

- The passed-in `_account` address is owned by the caller
- The account has not been activated on the same block that an owner is trying to close it on
- The account has no debt, i.e. has no active borrows

`closeAccount` will then transfer any remaining collateral back to the owner via the `sweepTo` function in `Account.sol`. `sweepTo` will iterate over the token collateral in an account and transfer the remaining balance to the passed-in address, which in this case is the account owner address. In addition to transferring any remaining token balances, `sweepTo` will also transfer any `ether` that was deposited as collateral by an account owner by calling the `safeTransferEth` function.

Transferring `ether` is reentrant in this use case because the owner of an account can be a contract that has either a `fallback() payable external` or `receive() payable external` method. Also, the `safeTransferEth` function uses the low-level `call` which is not given a `gas` parameter and thus 63/64ths of the remaining gas will be forwarded to the receiving address.

The handoff of execution flow as a result of transferring `ether` allows a malicious owner contract to reenter `closeAccount`. The reentrancy in `closeAccount` coupled with a lack of checks in critical state management functions gives a malicious owner contract the ability to perform an account takeover attack, which will now be explained in detail.

Moving back to the reentered `closeAccount` function, all of the initial checks that happen before the `sweepTo` call in `closeAccount` will pass. This includes the `onlyOwner` modifier that ensures only the account owner can call a particular method, since the logic that removes the owner address from an account occurs after the `sweepTo` function returns. The `sweepTo` function can then be called again, which will also succeed because there will be no asset tokens to transfer, and transferring 0 `ether` to an EOA or a contract is allowed. Once the second `sweepTo` call has finished, and the malicious owner contract has decided not to reenter `closeAccount` again, the subsequent call to the similarly-named `closeAccount` function in `Registry.sol` will occur. The `closeAccount` function in `Registry.sol` sets the owner address for the account that is being closed to the zero-address:

```
function closeAccount(address account) external accountManagerOnly {
    ownerFor[account] = address(0);
}
```

Note that there are no checks to ensure the owner for an account is not already the zero-address in the function above. In the final reentrant step, the `closeAccount` function will finish executing and will push the account address into the `inactiveAccounts` array. The original `closeAccount` function call will continue execution after the `safeTransferEth` function call and will perform the same action of calling `closeAccount` from `Registry.sol`. The account address will then be pushed into the `inactiveAccounts` array again. The final result of the malicious owner contract reentering `closeAccount` in `AccountManager.sol` is that the account address is now listed in the `inactiveAccounts` array twice.

The consequence of having the same account listed twice in the `inactiveAccounts` array is apparent when a new user goes to open an account via the `openAccount` function in `AccountManager.sol`:

```
function openAccount(address owner) external whenNotPaused {
    address account;
    if (inactiveAccounts.length == 0) {
        ...
    } else {
        account = inactiveAccounts[inactiveAccounts.length - 1];
        inactiveAccounts.pop();
        registry.updateAccount(account, owner);
    }
    IAccount(account).activate();
    emit AccountAssigned(account, owner);
}
```

In an effort to reduce gas costs associated with creating a new `Account` contract, inactive accounts will be used first until there are no more account addresses in the `inactiveAccounts` array. As part of initializing an inactive account, the `updateAccount` function in `Registry.sol` will set the owner for the account to the caller of `openAccount`:

```
function updateAccount(address account, address owner)
    external
    accountManagerOnly
{
    ownerFor[account] = owner;
}
```

Note that `updateAccount` does not have any checks to ensure the owner for the passed-in account address has not already been set. The `activate` function in `Account.sol` will be called to set the activation block of the account to the current block number.

Since there are no checks in `openAccount`, `updateAccount`, or `activate` to ensure an account must not already have an owner before transferring ownership, the next user who calls `openAccount` will receive the same account as the previous user, and the `AccountManager` will transfer ownership of the account from the previous user to the new user.

A malicious user could use the aforementioned reentrancy bug and account takeover vulnerability in the following attack scenario:

1. Alice deploys a malicious contract, which will be used to exploit the reentrancy bug in `closeAccount`.
2. Alice opens an account, passing in the contract address as the owner, transfers a small amount of `ether` to the account as collateral, and waits 1 block to ensure the `activationBlock` check in `closeAccount` succeeds.
3. Alice initiates the reentrancy by calling `closeAccount` via the malicious contract. `closeAccount` eventually transfers the `ether` to the malicious contract, which then calls `closeAccount` again. The reentered `closeAccount` call succeeds because the malicious contract is still the owner of the account, and then proceeds to call `closeAccount` in `Registry.sol` before finally pushing the now inactive account address into the `inactiveAccounts` array before exiting. The original `closeAccount` does the same steps, where `closeAccount` in `Registry.sol` also succeeds again because there are no checks to ensure an account's owner is not already the zero-address. The account address is now listed twice in the `inactiveAccounts` array.
4. A new user, Bob, goes to open an account. The last account address in `inactiveAccounts` is reused.
5. Bob decides to add collateral to his new account and deposits some `ether`.
6. Alice notices Bob has opened an account and back-runs his transaction that deposits `ether` into the account to ensure that she is the first to call `openAccount` again.
7. Alice calls `openAccount`, where the same account address is reused again because it was in the `inactiveAccounts` array twice, and ownership of the account is then transferred to Alice.
8. Alice then withdraws Bob's `ether` collateral and proceeds to exploit the `closeAccount` reentrancy bug again to setup the attack for a new victim.

RECOMMENDATION

The recommendation for this vulnerability is twofold. First, consider following the checks-effects-interactions pattern and removing a user as the owner of an account before calling `sweepTo` in the `closeAccount` function in `AccountManager.sol`. Second, due this critical vulnerability and another low vulnerability being found in the account reuse logic, consider redesigning the account reuse implementation in `AccountManager.sol`. If the intent is to optimize gas costs while still allowing users to own more than one account, consider using a mapping to link an owner address to one or more account addresses. For example, using a mapping of an owner address to array of `Account` contract addresses would still allow users to own multiple accounts, but the accounts would solely belong to one user and would not be repurposed for other users. Additionally, this would still allow users to repurpose their own closed accounts and reopen them again without having to pay for a new account to be created.

UPDATE

Fixed in pull request #192 (commit hash `9df8e61f582279ad2cf53f2c9699e1e4841bd7d5`), as recommended. Additionally, the Sentiment team added a new test to ensure the above attack is no longer possible.

MEDIUM SEVERITY

[M01] IMPRECISE INTEREST RATE CALCULATIONS

The `getBorrowRatePerBlock` function in `DefaultRateModel.sol` uses a non-linear interest rate model equation as the numerator, which is then divided by `blocksPerYear`. The `blocksPerYear` immutable variable represents the total number of blocks mined in a year and its default value is intended to be `2102400 * 1e18` as per a comment in `DefaultRateModel.sol`. However, using `blocksPerYear` as part of the interest rate calculation can lead to imprecise interest rates because the number of blocks mined per year is not constant and can be affected by a variety of different factors including:

- Consensus mechanism, i.e. Proof-of-Work block times vs. Proof-of-Stake block times
- Mining hashrate changes (Proof-of-Work)
- Varying network conditions of the specific blockchain the contract is deployed to

As a result, if `blocksPerYear` is too low, then the interest rate will be artificially higher per block which will end up costing borrowers more tokens, as a higher interest rate will be calculated over a larger number of blocks. If `blocksPerYear` is too high, then the interest rate will be lower per block, which means borrowers will end up paying less interest over a smaller number of blocks. In the former case the higher interest rate for borrowers benefits lenders, but in the latter case the lower interest rate for borrowers harms lenders as they will not be accruing as much interest for lending their tokens. Additionally, `blocksPerYear` is immutable so once set in the constructor, it cannot be modified.

RECOMMENDATION

Consider using seconds instead of blocks per year in `DefaultRateModel.sol`, and then calculate a time delta using `block.timestamp` instead of calculating a block delta using `block.number`.

UPDATE

Fixed in pull request #194 (commit hash `1a5494e512c2231732594f033fd3d619109075b3`), as recommended. The `blocksPerYear` storage variable was replaced with `secsPerYear`. One thing that should be noted is that `secsPerYear` can be marked constant and set using the `year` Solidity keyword.

[M02] POSSIBLE INTEGER UNDERFLOW WHEN CALCULATING INTEREST

> ***NOTE:*** *This issue was raised by the Sentiment team during the audit and was subsequently confirmed by AE.*

In the `LToken.sol` contract, the total `borrows` balance is calculated differently than each account's borrow balance. `borrows` is the total amount of underlying asset token that has been borrowed, plus interest. The interest for `borrows` is calculated by computing the current interest rate factor via the `getRateFactor` function, multiplying the interest rate by the current `borrows`, and adding that result back to `borrows`:

```
  uint rateFactor = getRateFactor();
  uint interestAccrued = borrows.mulWadUp(rateFactor);
  borrows += interestAccrued;
```

An account's borrow balance, which is stored in a `BorrowData` struct instance for each account address, is calculated using the `getBorrowBalance` function:

```
function getBorrowBalance(address account) public view returns (uint) {
    uint balance = borrowData[account].balance;
    return (balance == 0) ? 0 :
        (borrowIndex.mulWadUp(1e18 + getRateFactor()))
        .divWadDown(borrowData[account].index)
        .mulWadUp(balance);
}
```

`getBorrowBalance` uses the `borrowIndex` variable, which is a monotonically increasing value that represents the total interest that has been accumulated over all accounts. A ratio comprised of the current interest rate (obtained using `getRateFactor`) and an account's starting `index` is used to determine how much interest one account has accrued relative to how much it has borrowed.

Ideally, the summation of all account borrow balances should be equivalent to `borrows`; however, because of the rounding that will occur when calculating an account's borrow balance, this may not be the case. As a result, when a user is paying off the borrow balance owed to the `LToken` vault, the borrow balance may be larger than what was expected in `borrows`. In the case where the user is the only borrower this could lead to an integer underflow, and thus prevent a user from both fully paying off their borrow debt and closing their account.

RECOMMENDATION

Consider refactoring the interest calculations for `LToken` borrows to use the same calculations. The Sentiment team identified this as an issue independent of this audit and have a fix pending.

UPDATE

Fixed in pull request #189 (commit hash `574a5f0f9c5859ccc0a2e1bf17ceffe5b3ac7f52`), as recommended.

[M03] UNSAFE TRANSFER OF ERC-20 ASSETS

The `lendTo` function in `LToken.sol` uses the ERC-20 `transfer` function to send the underlying `LToken` ERC-20 asset to a user and will record the amount being lent as a borrow on the account:

```
function lendTo(address account, uint amt)
    ...
    asset.transfer(account, amt);
```

Multiple ERC-20 tokens return `false` instead of throwing on failure. If one of these ERC-20 tokens is used as the underlying asset of the `LToken`, it could result in a user having a borrow added to their account without them receiving the tokens they are entitled to because the return value of `transfer` is not checked. This would significantly impact a user as they would have to pay back the borrow (with interest) without receiving any portion of tokens.

RECOMMENDATION

Consider using either the `safeTransfer` function in `Helpers.sol` or the OpenZeppelin `safeTransfer` function. Both of these helper functions ensure that upon failure to transfer tokens, the entire transaction will revert.

UPDATE

Fixed in pull request #195 (commit hash `0c80307e0bb7ea51ac2c65ca00cd9b4a5490ca69`), as recommended.

LOW SEVERITY

[L01] APPROVALS NOT REMOVED DURING ACCOUNT CLOSURE

When a user closes a Sentiment account, the token approvals set by that user are not removed. As part of the account closure process, the Sentiment protocol adds closed accounts to a list of inactive accounts. When a user creates a new account, the Sentiment protocol will first try to assign that user an account from the set of inactive accounts. If the Sentiment protocol assigns an account with existing token approvals, then the new owner of the account will inherit those approvals.

The Sentiment team has expressed the possibility of allowing third-parties to write Controllers in the future. If this occurs, then accounts could be at risk of having their token balances spent by a malicious or vulnerable Controller if a previous owner of the account approved that Controller as a spender.

RECOMMENDATION

Due to this vulnerability and another critical vulnerability being found in account reuse, consider redesigning the implementation of account reuse in `AccountManager.sol`. If the intent is to optimize gas costs while still allowing users to own more than one account, consider using a mapping in either a one-to-many or many-to-one format. This would still allow users to own multiple accounts, but the accounts would solely belong to one user and would not be repurposed for other users. Additionally, this would still allow users to repurpose their own closed accounts and reopen them again without having to pay for a new account to be created. Since a user would only reuse accounts that once belonged to them, the persistence of old spender approvals is limited to approvals that the user themselves originally authorized.

UPDATE

Fixed in pull request #192 (commit hash `345e5590d5eda47b00e032c62f300dae6332d56a`), as recommended.

[L02] ERC-20 TOKENS ALLOW TRANSFER TO THE ZERO-ADDRESS

There are two ERC-20 tokens that inherit from the `solmate` contracts that are used in conjunction with the Sentiment Protocol to denote assets deposited into the protocol. The `LToken.sol` contract, the token issued for depositing ERC-20 assets, and `LEther.sol` contract, the token issued for depositing ETH into the protocol.

A common restriction on ERC-20 tokens is to disallow transfers to the zero-address. This helps to prevent accidental loss of tokens. However, the `LToken` and `LEther` tokens do not implement this safety check.

RECOMMENDATION

Consider adding a zero-address check when transferring ERC-20 tokens.

UPDATE

Fixed in pull request #196 (commit hash `78b4c6100e1e3c1896c62701e4930aae4bbdcfb9`), as recommended.

[L03] INCORRECT EVENT ARGUMENT

The Repay event is emitted as part of the _repay function. The event itself has 4 parameters as defined in the IAccountManager.sol interface:

```
event Repay(
    address indexed account,
    address indexed owner,
    address indexed token,
    uint value
);
```

Notice how the second indexed parameter for Repay is address indexed owner. In the _repay function, msg.sender is supplied as the argument for owner. msg.sender is correct when _repay is called by the repay function since that function can only be called by the owner of the account. However, any address can call the liquidate function, which calls _repay. In the case where an address that is not the owner calls liquidate, their address will be used as the argument to the owner parameter in the Repay event. Consequently, a user whose account has been liquidated may see via a UI or a block explorer one or more Repay events that show the liquidator as the owner, which is incorrect, and may cause confusion and/or concern for the user.

RECOMMENDATION

Consider using the ownerFor getter function in Registry.sol and replace msg.sender as the argument for owner in the Repay event with the address returned from the ownerFor function.

UPDATE

Fixed in pull request #193 (commit hash 7c5a5182ee0c002ad419275b3d2d2229a01ba227), as recommended. The Sentiment team removed the _repay function, which emitted the Repay event with the incorrect address.

[L04] MISSING CONTRACT ADDRESS CHECK

The following functions in the `Helpers.sol` contract wrap external function calls that will succeed if a contract does not exist at the address used to direct those function calls:

- `safeTransferFrom`
- `safeTransfer`
- `safeApprove`
- `withdraw`
- `safeApprove`

If the `address token` parameter can be manipulated to point to an EOA, the function call will succeed and it will not result in a reverted transaction as would be expected on a failure to transfer tokens.

If the `address token` parameter can be manipulated to point to an EOA, the corresponding `safe` function call will return `true` with no return data. These two conditions are sufficient for bypassing the `require` statements at the end of each function. As such, execution will continue and this will not result in a reverted transaction as would be expected on a failure to transfer tokens. Attackers have exploited similar circumstances to steal millions of US dollars worth of tokens.

RECOMMENDATION

Consider using OpenZeppelin's `SafeERC20.sol` contract, which provides functions to implement all the functions listed above.

Alternatively, consider merging the `isContract` function from OpenZeppelin into the existing `Helpers.sol` contract. This function will check whether the provided address points to a contract. Consider calling `isContract` before the wrapped function calls in the functions listed above and reverting when `isContract` returns `false`.

UPDATE

Fixed in pull request #197 (commit hash `16fcc631b399638820dd0518a59a5b51e3bfc20c`), as recommended. The `isContract` function was added to the following functions: * `safeTransferFrom` * `safeTransfer` * `safeApprove` (function signature of `safeApprove(address,address,uint256)`) * `withdraw`

It was not added to the following function: * `safeApprove` (function signature of `safeApprove(address,address,address,uint256)`)

This is acceptable because even if `safeApprove` is called on the 0 address, the call will succeed but the following call to `safeTransferFrom` will correctly revert on error.

[L05] MISSING SANITY CHECK ON OWNERSHIP TRANSFER

In the protocol's `Ownable.sol` contract, the `transferOwnership` [function](#) is used to update the administrator of the contract:

```
function transferOwnership(address newAdmin) external virtual adminOnly {
    emit OwnershipTransferred(admin, newAdmin);
    admin = newAdmin;
}
```

The function does not check that `newAdmin` is nonzero. If the current administrator transfers ownership to the zero-address, contracts will have to be redeployed to gain access to functions marked with the `adminOnly` modifier. These newly deployed contracts would then have to undergo a migration which would require substantial effort and cost.

It is less risky to use well-known library contracts for access control instead of creating your own. OpenZeppelin's Ownable contract contains similar functions and modifiers with additional safeguards.

RECOMMENDATION

Consider checking that `newAdmin` is nonzero, or prevent `transferOwnership` from setting `newAdmin` to an inaccessible address by splitting `transferOwnership` into a two step process, where the new administrator is required to claim ownership of the contract.

Furthermore, consider using OpenZeppelin's Ownable contract for access control.

This issue is identical to one AE identified in the previous audit of the Oracle and Controller contracts. As such, consider implementing the same fix across all versions of `Ownable.sol`.

UPDATE

Fixed in pull request #199 (commit hash `7c2e80b347e0dd69b248066e2916590ecdbbed3b`), as recommended. The Sentiment team added a check that ensures the `newAdmin` address is not zero.

[L06] PROXY INITIALIZATION REQUIRES MULTIPLE STEPS

The Sentiment Protocol uses transparent proxies to interact with implementation contracts and manage upgradeability. The proxies are deployed to the network, and then a second function call must be made to certain contracts in order to properly initialize state. This can be seen in how the `AccountManager` contract is deployed. Once `AccountManager` has been deployed, the `init` function must be executed which sets the address of the `registry`.

Due to the nature of mining and transaction ordering, a proxy can be upgraded and then initialized by a malicious front-running user if more than one transaction is used to upgrade.

RECOMMENDATION

Consider adding a function to the proxy that allows the implementation address to be modified and then execute a function on the new implementation address. OpenZeppelin has a reference implementation that provides this functionality. This code pattern ensures that as soon as a proxy is upgraded it will immediately be usable and not subject to multiple transaction ordering.

UPDATE

Acknowledged, and will implement as a future feature. The Sentiment team has an open pull request #218 that will add the `upgradeToAndCall` functionality. As this code has not been merged into `main` and is subject to change, the added functionality has not been audited.

[L07] RESERVE REDEMPTIONS CAN CAUSE SURPRISE INTEREST RATE CHANGES

The `_utilization` function in `DefaultRateModel.sol` calculates the utilization rate of an `LToken` vault by taking `borrows` and dividing that by `totalAssets`. `totalAssets` is made up of `borrows + liquidity` where `liquidity` is the current balance of the underlying asset token held by an `LToken` vault. Additionally, `liquidity` includes the `reserves` for an `LToken` vault as there is no delineation present that would restrict the `LToken` vault from lending out its entire balance of asset tokens when `reserves` is nonzero. However, the admin of the given `LToken` vault has the ability to call the `redeemReserves` function in `LToken.sol`:

```
function redeemReserves(uint amt) external adminOnly {
    updateState();
    reserves -= amt;
    emit ReservesRedeemed(treasury, amt);
    asset.transfer(treasury, amt);
}
```

`redeemReserves` withdraws up to `amt` from `reserves` and transfers the tokens to the `treasury` address. The Sentiment documentation gives more context on when `redeemReserves` may be called:

> *[`redeemReserves`] Transfers reserves from the LP to the specified address, this will be used sparsely, when assets accrued via the reserve factor are necessary for utilization. This could be used for a liquidity backstop or fee disbursement.*

However, `redeemReserves` can be used by the admin of a given `LToken` vault to influence the utilization rate, which ultimately influences the interest rate for the `LToken` vault. Depending on the value of `reserves` and the amount being redeemed, this can have a non-negligible impact on the interest rate calculation of an `LToken` vault. The higher interest rate would then impact all users who are currently borrowing from the `LToken` vault. Therefore, the act of redeeming `reserves` for the treasury could be misconstrued as manipulation and generate ill will.

RECOMMENDATION

Consider implementing a timelock so that users have advance notification of `reserves` being redeemed from an `LToken` vault.

UPDATE

Fixed via implementing a timelock on a multi-sig wallet that controls reserve redemptions. Statement from the Sentiment team:

> *Timelock will be implemented on the controlling multi-sig wallet*

[L08] UNNECESSARY `RECEIVE` FUNCTION

The `receive() external payable {}` function in `AccountManager.sol` is unnecessary. The `AccountManager` contract does not hold user's funds, and if a user wishes to deposit `ether` into their account they should use the `depositEth` function in `AccountManager.sol` instead. Having this unnecessary `receive` could cause users to mistakenly send `ether` to the `AccountManager` contract. This would cause a user to lose their `ether` permanently because `AccountManager.sol` does not have a recovery mechanism.

RECOMMENDATION

Consider removing the `receive` function from `AccountManager.sol`.

UPDATE

Fixed in pull request #198 (commit hash `8322656631a974496c5daa7e56078fe070c3f74d`), as recommended.

NOTE SEVERITY

[N01] APPROVAL RACE CONDITION

The `safeApprove` function in the `Helpers.sol` contract uses the ERC-20 `approve` function to set spender approvals:

```
function safeApprove(address account, address token, address spender, uint value)
internal {
    (bool success, bytes memory data) = IAccount(account).exec(token, 0,
        abi.encodeWithSelector(IERC20.approve.selector, spender, value));
    require(success && (data.length == 0 || abi.decode(data, (bool))),
"APPROVE_FAILED");
}
```

The `approve` function is vulnerable to a race condition when an address grants an allowance more than once to an address. If a second allowance for a spender is set via the `approve` function, the spender with the allowance can front run the second approval and transfer tokens using the first allowance. After the second allowance is set, the spender can then spend that allowance. The result is that the spender can withdraw more tokens from the address than intended. This is a known vulnerability.

The Sentiment protocol only grants allowances to Controllers. However, the Sentiment team has expressed the possibility of allowing third-parties to write Controllers. A malicious, third-party controller could perform the spending attack described above.

RECOMMENDATION

Consider using OpenZeppelin's implementation of the `safeIncreaseAllowance` function and `safeDecreaseAllowance` function to set and change spending approvals.

UPDATE

Acknowledged, and will not fix. Sentiment's statement on the issue:

> *Fix not implemented since Sentiment does not expect third-party controllers in the near future.*

[N02] CONTRACTS USE FLOATING COMPILER VERSION PRAGMA

All Sentiment Protocol contracts float their Solidity compiler versions (e.g. `pragma solidity ^0.8.10`).

Locking the compiler version prevents accidentally deploying the contracts with a different version than what was used for testing. The current pragma prevents contracts from being deployed with an outdated compiler version, but still allows contracts to be deployed with newer compiler versions that may have higher risks of undiscovered bugs.

It is best practice to deploy contracts with the same compiler version that is used during testing and development.

RECOMMENDATION

Consider locking the compiler pragma to the specific version of the Solidity compiler used throughout the application's lifecycle.

UPDATE

Fixed in pull request [#200](#) (commit hash `9e196f3aa32a39a2921710c733167b26b554e2d8`), as recommended. The Solidity compiler version chosen was 0.8.15.

[N03] DUPLICATE FUNCTION NAMES

The `AccountManager.sol`, `Account.sol`, and `Helpers.sol` contracts contain duplicate function names:

- The `exec` function in `AccountManager.sol` and `Account.sol`
- The `withdrawEth` function in `AccountManager.sol` and `Helpers.sol`
- The `withdraw` function in `AccountManager.sol` and `Helpers.sol`

Duplicate function names can make understanding the protocol more difficult for users and other developers.

RECOMMENDATION

Consider renaming functions with the same name to have unique names.

UPDATE

Acknowledged, and will not fix. Sentiment's statement on the issue:

> *It was a conscious design decision to have matching function names. Trying to maintain uniformity in particular flows i.e. "AccountManager.exec calls Account.exec" or "AccountManager.withdraw calls Helpers.sol"*

[N04] EVENT PARAMETER MISSING `INDEXED` KEYWORD

The `Upgraded` event in `Proxy.sol` does not index the `address` parameter. This is inconsistent with EIP-1967. This could lead to errors when parsing event data in locations that expect the address to be indexed.

RECOMMENDATION

Consider indexing the `address` parameter of the `Upgraded` event.

UPDATE

Fixed in pull request #201 (commit hash `37b9780c36bdaf508146e7f64e2184bc5afac6fb`), as recommended.

[N05] EXTRANEOUS `PAYABLE` KEYWORD ON `EXEC` FUNCTION

The `exec` function in `Account.sol` has the `payable` keyword; however, it is not possible to send `ether` to this function. Only `AccountManager.sol` is allowed to call `exec` in `Account.sol` via its own similarly-named `exec` function, but that function does not have the `payable` keyword. As such, if `msg.value` is non-zero when the `exec` function is called in `AccountManager.sol`, the transaction will revert.

RECOMMENDATION

Consider marking the `exec` function in `AccountManager.sol` as `payable` if the intent is to allow `ether` to be sent to the `exec` function in `Account.sol`. Otherwise consider removing the `payable` keyword from the `exec` function in `Account.sol`.

UPDATE

Fixed in pull request #202 (commit hash `982704c549a7607d3c9276ec4476b1a73a349cc3`), as recommended.

[N06] HARDCODED ADDRESS FOR AAVE LENDINGPOOL

The `AaveV2.t.sol` unit tests hardcode the address of the Aave LendingPool contract. Additionally, the documentation in `KovanDeployment.md` lists a hardcoded address for the LendingPool contract.

The Aave LendingPool contract handles all user-oriented actions in the Aave protocol, which makes using the current version of the contract important for security purposes. If Aave were to redeploy this contract because of a security vulnerability before a redeployment of the Sentiment Protocol contracts, and the hardcoded LendingPool address in the Sentiment redeployment code is not modified, then the Sentiment Protocol would use the vulnerable LendingPool contract.

RECOMMENDATION

Consider using the `getLendingPool` function on Aave's `LendingPoolAddressesProvider.sol` contract to retrieve the address of the Aave Lending Pool. Additionally, consider updating unit tests and documentation to reflect that the protocol retrieves the LendingPool contract address in this manner.

UPDATE

Fixed in pull request #204 (commit hash `4a237e44bd4ae65955aae56080b687c67aaad311`), as recommended.

[N07] INCOMPLETE RESETTING OF STATE WHEN ACCOUNT IS CLOSED

The `activationBlock` state variable in `Account.sol` is described by a comment:

```
/// @notice Block number for when the account is activated
```

However, while the `activationBlock` storage variable may be set as part of the `activate` function in `Account.sol`, `activationBlock` is not reset to 0 when an account is closed. This means `activationBlock` will always return a non-zero value even when an account is inactive, which may confuse users.

RECOMMENDATION

Consider adding logic to set `activationBlock` to 0 when an account is closed.

UPDATE

Fixed in pull request #210 (commit hash `54cb5b66b769f244ef2ef2d158aa4142e49197f6`), as recommended.

[N08] INCORRECT NATSPEC DOCUMENTATION

Several functions throughout the Sentiment Protocol such as `collectFrom` in `LToken.sol` reference variables that do not exist in the function.

```
/**
@notice Collects a specified amount of underlying asset from an account
@param account Address of account
@param amt Amount of token to collect
@return isNotInDebt Returns if the account has pending borrows or not
*/
function collectFrom(address account, uint amt)
...
    return (borrowData[account].balance == 0);
```

In the above NatSpec comment, `isNotInDebt` is stated as the name of the return variable, but `isNotInDebt` is not defined in the function.

RECOMMENDATION

Consider removing variable names that do not exist in the function body from NatSpec function comments.

UPDATE

Partially fixed in pull request #203 (commit hash `97f00a417dce56e1c4f4ab85531c64bcc93ed661`). The issue in the example above was fixed, however other instances of the issue still persist throughout the codebase.

[N09] INCORRECT ARGUMENT TYPE

The `Account.sol` contract defines the `exec` function with an argument `data` of type `bytes calldata`:

```
function exec(address target, uint amt, bytes calldata data)
    external
    payable
    accountManagerOnly
    returns (bool, bytes memory)
{
    (bool success, bytes memory retData) = target.call{value: amt}(data);
    return (success, retData);
}
```

The `IAccount.sol` contract declares the same `exec` function with the argument `data` of type `bytes memory`:

```
function exec(
    address target,
    uint amt,
    bytes memory data
) payable external returns (bool, bytes memory);
```

RECOMMENDATION

Consider updating the declaration of `exec` in `IAccount.sol` to use the `bytes calldata` type for the `data` argument.

UPDATE

Fixed in pull request #211 (commit hash `71cc875b89fdffa83fe68f5a2a6ca4af9e0af8f3`), as recommended.

[N10] INCORRECT COMMENT ABOUT UTILIZATION RATE

The NatSpec comment for the `getBorrowRatePerBlock` function in `DefaultRateModel.sol` references an outdated utilization rate equation. The equation was changed as of commit `a03654bf88a888fe5865405bdb21d4ac27a5eae2` to reflect `reserves` being a part of the overall `liquidity` for a given `LToken` vault.

RECOMMENDATION

Consider updating the comment to reflect the new utilization rate equation.

UPDATE

Fixed in pull request #203 (commit hash `4d691bb110c17f5bfe8c78a861966f2400662417`), as recommended.

[N11] MAGIC NUMBER IN `LTOKEN.SOL`

The number `1e18` is used in multiple calculations in `LToken.sol`; however, its use varies per function. The lack of comments makes the reason for each usage difficult to understand.

For example, in the `getRateFactor` function `1e18` is used to scale `blockDelta` to 18 decimals, but in the `getBorrowBalance` function `1e18` is used to ensure the `balance` will be non-zero even if `getRateFactor` returns 0.

RECOMMENDATION

Consider adding a comment to each use of `1e18` to explain why it is necessary for the equation.

UPDATE

Fixed in pull request #213 (commit hash `8e09a3e06a329776ec075b654afc7c40d6bfb0ce`), as recommended.

[N12] MISLEADING USE OF FUNCTION PARAMETER NAMES

Throughout the codebase, the function parameters `value` and `amt` are used interchangeably, which is incorrect and misleading. Both are used in relation to tokens; however, `value` should be used to represent the value of a token and `amt` should be used to represent token balance. As such, using the two interchangeably can lead to confusion for other developers and users who may be viewing the codebase.

RECOMMENDATION

Consider replacing the parameter names `value` and `amt` with the appropriate name that is applicable to the context that the variable is being used in throughout the codebase.

UPDATE

Fixed in pull request #220 (commit hash `866cf8430fa9f85713f709cce32a6333c31e1db8`) and pull request #213 (commit hash `037770e558fb6981b763933aabcdaa0954bc086e`), as recommended.

[N13] MISSING `VIEW` FUNCTION MODIFIER

The `IRiskEngine.sol` interface contract defines the following functions without the `view` function modifier. However, the actual function implementations in the `RiskEngine.sol` contract are marked `view`:

- `getBorrows`
- `isAccountHealthy`
- `isBorrowAllowed`
- `isWithdrawAllowed`

Interfaces should match function implementations for ease of understanding.

RECOMMENDATION

Consider adding the `view` function modifier to the above listed functions in `IRiskEngine.sol`.

UPDATE

Fixed in pull request #209 (commit hash `ba1ac4d41449b3ae8e301fa38f8862d53a736890`), as recommended.

[N14] MISSING CONSTRUCTOR SANITY CHECKS

The immutable variables `c1`, `c2`, `c3`, and `blocksPerYear` are set in the `constructor` [function](#) of `DefaultRateModel.sol`:

```
/**
    @notice Contract constructor
    @param _c1 constant coefficient, default value = 1 * 1e17
    @param _c2 constant coefficient, default value = 3 * 1e17
    @param _c3 constant coefficient, default value = 35 * 1e17
    @param _blocksPerYear blocks in a year, default value = 2102400 * 1e18
*/
constructor(uint _c1, uint _c2, uint _c3, uint _blocksPerYear) {
    c1 = _c1;
    c2 = _c2;
    c3 = _c3;
    blocksPerYear = _blocksPerYear;
}
```

While [the code comments](#) suggest appropriate default values, the constructor lacks sanity checks to prevent setting the values to 0 or to values that are off by an order of magnitude.

RECOMMENDATION

Consider adding sanity checks to the `constructor` in `DefaultRateModel.sol` to guard against setting any immutable variables to 0 or to an incorrect magnitude.

UPDATE

Partially fixed in pull request [#206](#) (commit hash `e06854d27a63aca07562a9b8ab5915faa93e9042`). The Sentiment team added a check against null values, but did not add a check against values of an incorrect magnitude.

[N15] MISSING INITIALIZATION ZERO-ADDRESS CHECK

The `LToken.sol` contract has an `init` function that is called by a proxy to initialize the state. The `init` function does not check the `_asset` address parameter to ensure it is not equal to the zero-address. If the proxy accidentally sets `_asset` to the zero-address when calling `init`, the proxy will have to be redeployed.

RECOMMENDATION

Consider adding a check in the underlying `ERC4626.sol` contract to ensure `_asset` is not the zero-address when executing the `init` function.

UPDATE

Fixed in pull request #208 (commit hash `3643dfdd4ebdfb5c7d97411e8661bc3a76887f4d`), as recommended.

[N16] MIXED COMPILER PRAGMA DIRECTIVES

The majority of the contracts in the Sentiment Protocol repository target version `0.8.10` or later of the Solidity compiler. The following contracts break this convention:

- `IERC4626.sol`
- `Ownable.sol`

Using different pragma directives can leave contracts susceptible to different sets of compiler bugs.

RECOMMENDATION

Consider making the Solidity version consistent across all contracts within the project.

UPDATE

Fixed in pull request #200 (commit hash `80b37b61075220faedfbe9554e999e49775f7497`), as recommended.

[N17] MIXED USAGE OF FIXED-POINT ARITHMETIC LIBRARIES

The Sentiment Protocol contracts make use of both `PRB-Math` and Solmate's `FixedPointMathLib` fixed-point arithmetic libraries. Using different libraries to perform calculations can lead to issues such as unexpected rounding errors.

RECOMMENDATION

Consider using only one fixed-point arithmetic library.

UPDATE

Fixed in pull request [#214](#) (commit hash `14af006a2f3e4de58bcef574e40a408315c49984`), as recommended.

[N18] NOT USING NORMALIZED NOTATION FOR `BALANCETOBORROWTHRESHOLD`

In the `RiskEngine.sol` [contract](#) the number `1.2` is stated as the ratio of balance to borrow:

```
/// @notice Balance:Borrow, Default = 1.2
uint public constant balanceToBorrowThreshold = 12 * 1e17;
```

The ratio has 18 decimals but `12 * 1e17` is used which makes understanding the purpose and correct usage of `balanceToBorrowThreshold` difficult.

RECOMMENDATION

Consider defining the storage variable using `1e18` as the base multiplier. Declaring `1.2 * 1e18` is [valid syntax](#) and increases code clarity.

UPDATE

Fixed in pull request [#207](#) (commit hash `aa01e648c5200f5689ef03277d8a06c8895c3e7e`), as recommended.

[N19] UNUSED ERRORS

The `Errors.sol` contract contains the following unused errors:

- `AccountsNotFound`
- `PriceFeedUnavailable`

RECOMMENDATION

Consider removing the errors listed above from `Errors.sol`.

UPDATE

Fixed in pull request #205 (commit hash `345f0efd90b447fc5cad27feac69c225b7377ac1`), as recommended.

[N20] UNUSED FUNCTIONS

The `Helpers.sol` contract contains the following unused functions:

- safeApprove
- isEth

RECOMMENDATION

Consider removing the functions listed above from `Helpers.sol`.

UPDATE

Fixed in pull request #205 (commit hash `345f0efd90b447fc5cad27feac69c225b7377ac1`), as recommended.

[N21] UNUSED INTERFACE

The `IBeaconProxy.sol` contract is not used by any other files in the protocol repository. Additionally, the single interface defined in that contract, `IBeaconProxy`, does not have a matching function in the `BeaconProxy.sol` contract.

RECOMMENDATION

Consider removing the `IBeaconProxy.sol` file.

UPDATE

Fixed in pull request #205 (commit hash `345f0efd90b447fc5cad27feac69c225b7377ac1`), as recommended.

[N22] UNUSED STORAGE VARIABLE

The `LToken.sol` contract defines a storage variable `borrowFeeRate`:

```
/// @notice unused
uint public borrowFeeRate;
```

`borrowFeeRate` is not used in the `LToken.sol` contract (and is noted as such in a comment).

RECOMMENDATION

Consider removing the `borrowFeeRate` storage variable.

UPDATE

Fixed in pull request #189 (commit hash `2fd383a0ae5949e84b1a375abd2a9e39ea32f168`), as recommended.

[N23] ZERO-ADDRESS OWNED ACCOUNTS

The `openAccount` function allows users to create accounts owned by the zero-address. To mark an account as inactive after the account is closed, the protocol calls the `closeAccount` function, which sets the owner of the account to the zero-address:

```
/**
    @notice Closes account
    @dev Sets address of owner for the account to 0x0
    @param account Address of account to close
*/
function closeAccount(address account) external accountManagerOnly {
    ownerFor[account] = address(0);
}
```

RECOMMENDATION

Consider updating `openAccount` to revert if the `owner` argument is the zero-address.

UPDATE

Fixed in pull request #219 (commit hash `ae21d97ebd5a9ec4835fda74b921174af436f5b3`), as recommended.

[N24] ZERO-ADDRESS RETURNED DURING CONTRACT NAME LOOKUP

The `addressFor` mapping in `Registry.sol` maps a contract name as a string (e.g. "REGISTRY", "ACCOUNT_MANAGER", etc.) to the address at which the contract is deployed. The address retrieval logic does not check whether `addressFor` returns the zero-address. The `addressFor` mapping could return the zero-address in the case of a contract name misspelling or a deployment issue (e.g. forgetting to update the `addressFor` mapping after a new Sentiment contract is deployed). In the event that `addressFor` does return the zero-address, the protocol will treat that address as a legitimate contract address and attempt to execute functions at the zero-address.

RECOMMENDATION

Consider adding a `getAddress` function (as a counterpart to the `setAddress` function), which queries the `addressFor` mapping and reverts if the zero-address is returned.

UPDATE

Fixed in pull request #212 (commit hash `1988cff7b8546bb02fab29fb3003e5f03c153493`), as recommended.

APPENDIX

APPENDIX A: SEVERITY DEFINITIONS

Severity	Definition
Critical	This issue is straightforward to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
High	This issue is difficult to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
Medium	This issue is important to fix and puts a subset of users' data at risk and is possible to lead to moderate financial impact.
Low	This issue is not exploitable in a recurring basis and cannot have a significant impact on execution.
Note	This issue does not pose an immediate risk but is relevant to security best practices.

APPENDIX B: FILES IN SCOPE

```
src/core/Account.sol
src/core/AccountFactory.sol
src/core/AccountManager.sol
src/core/DefaultRateModel.sol
src/core/Registry.sol
src/core/RiskEngine.sol
src/interface/core/IAccount.sol
src/interface/core/IAccountFactory.sol
src/interface/core/IAccountManager.sol
src/interface/core/IRateModel.sol
src/interface/core/IRegistry.sol
src/interface/core/IRiskEngine.sol
src/interface/proxy/IBeacon.sol
src/interface/proxy/IBeaconProxy.sol
src/interface/tokens/IERC20.sol
src/interface/tokens/IERC4626.sol
src/interface/tokens/ILEther.sol
src/interface/tokens/ILToken.sol
src/interface/utils/IOwnable.sol
src/proxy/BaseProxy.sol
src/proxy/Beacon.sol
src/proxy/BeaconProxy.sol
src/proxy/Proxy.sol
src/tokens/LEther.sol
src/tokens/LToken.sol
src/tokens/utils/ERC20.sol
src/tokens/utils/ERC4626.sol
src/utils/Errors.sol
src/utils/Helpers.sol
src/utils/Ownable.sol
src/utils/Pausable.sol
src/utils/Storage.sol
```