Term Finance

Term Finance – Smart Contract Changes

Security Assessment Report

Version: 2.0

May, 2023

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of a selected list of changes made to the Term Finance smart contracts.

The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Term Finance smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Term Finance smart contracts.

Overview

Term Finance is a noncustodial fixed-rate liquidity protocol modeled on tri-party repo arrangements common in traditional finance.

Liquidity suppliers and takers are matched through a unique weekly auction process where liquidity takers submit bids and suppliers submit offers to the protocol, which then determines an interest rate that clears the market.

Bidders who bid more than the clearing rate receive liquidity and lenders asking less than the clearing rate, supply.

Security Assessment Summary

This review was conducted on the files hosted on the term-finance repository, with majority of code changes assessed at commit ba9550e.

The scope of this assessment was strictly limited to the code changes related to the following PRs:

- PR 762
- PR 764
- PR 766
- PR 769
- PR 772
- PR 775
- PR 779
- PR 782
- PR 784
- PR 802
- PR 816
- PR 817, PR 818
- PR 820, PR 828, PR 830
- PR 822
- PR 825
- PR 834, PR 837
- PR 850 (relevant changes reviewed at commit 5d2251f)
- PR 853 (relevant changes reviewed at commit c112f92)
- PR 856 (relevant changes reviewed at commit f11982e)
- PR 857 (relevant changes reviewed at commit b6da64c)

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contract changes in scope. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 5 issues during this assessment. Categorized by their severity:

- Critical: 1 issue.
- High: 2 issues.
- Medium: 1 issue.
- Informational: 1 issue.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the code changes made to Term Finance's smart contracts, as per the scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
TRM2-01	Duplicate Bid and Offer IDs	Critical	Resolved
TRM2-02	`encumberedCollateralBalances` Not Updated	High	Closed
TRM2-03	Auctions Require Too Much Gas to Complete	High	Open
TRM2-04	Bid and Auction Limits Allow Participants to Lock Out All Competition	Medium	Closed
TRM2-05	Miscellaneous General Comments	Informational	Closed

TRM2-01	Duplicate Bid and Offer IDs		
Asset	`TermAuctionBidLocker.sol`, `TermAuctionOfferLocker.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Users can overwrite their existing bids, stranding collateral and making auction resolution impossible.

When a new bid is submitted with a bid ID that does not already exist, a new bid ID is generated and used. However, this new bid ID is determined by the bid submission, and so the same submission always creates the same bid ID. This makes it possible to overwrite existing bids for the same user. When this happens, the tally of bids is incorrectly increased and this makes it impossible to either complete or cancel the auction. The user is also unable to unlock their collateral from the first submission.

The logic for determining the bid ID of a bid submission is contained in `_lock()`:

```
447 bool bidExists = bids[bidSubmission.id].amount != 0;
    bytes32 bidId;
449 if (bidExists) {
        if (bids[bidSubmission.id].bidder != bidSubmission.bidder) {
451         revert BidNotOwned();
        }
453     bidId = bidSubmission.id;
    } else {
455     bidId = _generateBidId(bidSubmission.id, authedUser);
    }
```

```
968 function _generateBidId(
        bytes32 id,
970     address user
    ) internal view returns (bytes32) {
972     return keccak256(abi.encodePacked(id, user, address(this)));
    }
```

As can be seen, if a submission is made with an existing bid ID, the variable `bidExists` is set to `True` and the code proceeds to modify the existing bid. However, if the submission contains a bid ID which does not exist, it deterministically generates a new bid ID based on only the user's address and the submitted bid ID. Critically, this generated bid ID is not checked against existing bid IDs.

Because of this, it is possible to generate the same bid ID multiple times. Each time, the variable `bidExists` will be set to `False`, however, and so the bid submission will be treated as a new bid. It will overwrite the existing bid's information, transfer new collateral without regard for existing collateral, and increment `bidCount`, the contract's counter for the number of submitted bids.

This last point will make it impossible to either complete or cancel the auction as both of the relevant functions in `TermAuction`, `completeAuction()` and `cancelAuction()`, call `getAllBids()` in `TermAuctionBidLocker`. This function loops through the submitted bids, processes them, and, crucially, decrements `bidCount`. If `bidCount` is not zero after this loop, it reverts. This revert will always occur in the situation where the number of bids is less than `bidCount`, as is the case here.

In addition, as the functions that unlock bids and return collateral rely on the values stored in `bids`, and these have been overwritten, it will not be possible for the user to unlock the collateral from their overwritten bids.

This issue also applies to `TermAuctionOfferLocker`, which uses similar code.

Recommendations

Change the logic of `_generateBidId()` to check that the generated ID does not exist and only return once a unique bid ID has been generated.

Resolution

This finding has been resolved in PR 868.

TRM2-02	`encumberedCollateralBalances` Not Updated		
Asset	`TermRepoCollateralManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

`auctionLockCollateral()` and `auctionUnlockCollateral()` do not modify `encumberedCollateralBalances` array, resulting in collateral locked or unlocked by the auction process not being tracked.

This may result in an invalid accounting, causing unexpected issues further down in the execution flow.

Recommendations

Ensure the ledgers are correctly updated from within `auctionLockCollateral()` and `auctionUnlockCollateral()` functions, for example:

```
// for locking
lockedCollateralLedger[borrower][collateralToken] += amount;
encumberedCollateralBalances[collateralToken] += amount;
```

Alternatively, call `_lockCollateral()` and `_unlockCollateral` functions from within `auctionLockCollateral()` and `auctionUnlockCollateral()` respectively (the same way as it is currently implemented for `externalLockCollateral()` and `externalUnlockCollateral()`).

Resolution

This finding has been closed as false-positive. Upon consultation with the development team, this is an expected behaviour - collateral is only considered encumbered when a loan is secured, which is handled by `fulfillBid()`.

TRM2-03	Auctions Require Too Much Gas to Complete		
Asset	`TermAuction.sol`, `TermAuctionBidLocker.sol`, `TermAuctionOfferLocker.sol`		
Status	**Open**		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The gas usage of the `TermAuction` function `completeAuction()` uses approximately 1.5m gas for every 10 bids and offers in the auction. It is therefore estimated to reach the block gas limit at approximately 200 bids and offers.

Note that, at a gas price of 100 wei, the block gas limit of 30m gas would cost 3 ether.

The current settings of `MAX_BID_COUNT` and `MAX_OFFER_COUNT` in `TermAuctionBidLocker` and `TermAuctionOfferLocker` are both 1,000. The block gas limit would be reached significantly before this limit, and therefore they would not have their stated effect of preventing the block gas limit from preventing auction completion.

Recommendations

Consider changing the auction resolution logic to be significantly more gas efficient. Alternatively, set these limits to 180.

Resolution

TBD.

TRM2-04	Bid and Auction Limits Allow Participants to Lock Out All Competition		
Asset	`TermAuction.sol`, `TermAuctionBidLocker.sol`, `TermAuctionOfferLocker.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The current settings of `MAX_BID_COUNT` and `MAX_OFFER_COUNT` in `TermAuctionBidLocker` and `TermAuctionOfferLocker` prevent bids and offers past a certain count. In the current code, these values need to be relatively low: they are both set to 1,000 (although may need to be lower: see TRM2-03). It would be possible for an auction participant to submit many tiny bids in a single transaction and thus lock out all other participants.

One bidder could call `TermAuctionBidLocker::lockBids()` multiple times in a single transaction to create 999 tiny bids and one large bid, all at the minimum level. At this point, only one bidder exists, but no others can enter the auction.

This issue is mitigated by the fact that the offerers could withdraw their offers or the auction could be cancelled.

Recommendations

Consider changing the auction resolution logic to be significantly more gas efficient so that these limits are not needed.

Resolution

This finding has been risk accepted with the following advice from the development team:

Unfortunately we do not have the bandwidth for gas optimizations (these would be major changes to our conctracts), this is something we will look to address in the the next iteration. Currently we address this vulnerability by setting a minimum tender amount (See line [485]) to prevent a user from flooding the auction with low value tenders to block out other users.

TRM2-05	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Underflow risk in `TermRepoServicer`.**

   The below code will underflow if `totalOutstandingRepurchaseExposure < repurchaseExposureLedger[borrower]`.

```
655 if (rolloverSettlementAmount > repurchaseExposureLedger[borrower]) {
        totalOutstandingRepurchaseExposure -= repurchaseExposureLedger[
657         borrower
        ];
```

   Implement checks to ensure `totalOutstandingRepurchaseExposure` is always greater than or equal to `repurchaseExposureLedger[borrower]`.

   Note, this did not appear to be exploitable at the time of the review due to the relevant values being updated together throughout the `TermRepoServicer.sol` contract.

2. **`TermAuctionOfferLocker::unlockOfferPartial()` and `TermAuctionBidLocker::auctionUnlockBid()` do not check if a bid or an offer exists.**

   Note, this did not appear to be exploitable at the time of the review due to the auction contract using `getAllBids()` and `getAllOffers()` before calling `unlockOfferPartial()` or `auctionUnlockBid()` functions, which already check for the existence of bids and offers.

3. **Return new IDs from `TermAuctionOfferLocker::lockOffer()` and `TermAuctionBidLocker::lockBid()` to increase composability**

   These functions to lock bids and offers will often modify the submitted bid and offer IDs. However, the only way to access this information is through the event emitted. It would make the protocol easier to interact with if there were an accessible on chain method for finding out the generated bid or offer ID. Consider returning generated IDs from these and similar functions.

4. **Code quality and consistency improvements - `TermAuctionOfferLocker`**

   To check if an offer exists, there is currently `onlyExistingOffer(id)` modifier added to `unlockOffer()` function, and a separate check implemented for `unlockOffers()`.

   For consistency and readability, move the following check to `_unlock()` function (the same way as it is implemented in `TermAuctionBidLocker`):

```
if (offers[offerSubmission.id].amount == 0) {
    revert NonExistentOffer(offerSubmission.id);
}
```

5. **Typo**

   "aalready" on line [324] of `TermRepoCollateralManager`

6. **Redundant code**

   On line [326] of `TermRepoCollateralManager`, `decrementEncumberedCollateral` is conditionally set to `false`. However, this variable was declared on line [322] and would default to `false`. This first part of the test therefore performs no function. Consider replacing it with a comment to preserve clarity.

```
322  bool decrementEncumberedCollateral;

324  // collateral has aalready been unencumbered through liquidation
     if (termRepoServicer.getBorrowerRepurchaseObligation(borrower) == 0) {
326    decrementEncumberedCollateral = false;
     } else {
328    decrementEncumberedCollateral = true;
     }
```

7. **Inconsistent use of `storage` variable**

   On line [363] of `TermRepoRolloverManager`, a variable `rolloverElection` is set as the entry in the state variable `rolloverElections` for the account being rolled over. This is set as a `storage` variable. However, the original state variable, `rolloverElections[borrowerToRollover]` is still referenced in the code on line [371], line [376], line [379], line [427] and line [436]. Creating a `storage` variable for it therefore just results in the code being less usable.

   Additionally, there are no gas savings, as `rolloverElection` is a `storage` variable and so reading from it will require storage reads. Consider, therefore, declaring it as a `memory` variable.

8. **Event names**

   In `TermAuction::AuctionCompleted()`, the variables and event entries `totalAssignedBids` and `totalAssignedOffers` sound like the counts of bids and offers but are actually total amounts. Consider renaming them to put the word "total" at the end to emphasise that they are totals: `assignedBidsTotal` and `assignedOffersTotal`.

9. **PR 850: revert before operations to save gas**

   On line [442] of `TermRepoCollateralManager`, the variable `totalClosureRepoTokenAmounts` is tested and there is a revert if it equals zero. This could be done before `_transferLiquidationCollateral()` is called in the preceding lines. That function does not modify `totalClosureRepoTokenAmounts` and so it will save gas if a transaction reverts at this point.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The comments above have been acknowledged by the development team and selected findings were actioned in the following PRs:

- PR 873
- PR 874
- PR 876

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `brownie` framework was used to perform these tests and the output is given below.

```
============================================================ test session starts =============================================================
platform linux -- Python 3.8.12, pytest-6.2.5, py-1.10.0, pluggy-1.0.0
plugins: eth-brownie-1.17.1, mock-3.7.0, web3-5.24.0, cov-3.0.0, hypothesis-6.24.0, xdist-1.34.0, forked-1.3.0
collected 99 items

Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 20 --hardfork istanbul --mnemonic brownie --defaultBalanceEther
      ↪  1000000'...

tests/test_Authenticator.py ...xx.                                                                                                      [  6%]
tests/test_TermAuction.py .....                                                                                                         [ 11%]
tests/test_TermAuctionBidLocker.py .......s.....                                                                                        [ 31%]
tests/test_TermAuctionOfferLocker.py ...s.............                                                                                  [ 49%]
tests/test_TermController.py ...                                                                                                        [ 52%]
tests/test_TermEventEmitter.py .                                                                                                        [ 53%]
tests/test_TermInitializer.py ..                                                                                                        [ 55%]
tests/test_TermPriceConsumerV3.py .                                                                                                     [ 56%]
tests/test_TermRepoCollateralManager.py .x.......                                                                                       [ 65%]
tests/test_TermRepoLocker.py ...                                                                                                        [ 68%]
tests/test_TermRepoRolloverManager.py ...........                                                                                       [ 79%]
tests/test_TermRepoServicer.py .......                                                                                                  [ 86%]
tests/test_TermRepoToken.py .........                                                                                                   [ 95%]
tests/test_poc_duplicate-bidid.py .                                                                                                     [ 96%]
tests/test_poc_max_price.py .                                                                                                           [ 97%]
tests/test_poc_multicollateral.py x                                                                                                     [ 98%]
tests/test_poc_revert_auth.py x                                                                                                         [100%]

============================== 92 passed, 2 skipped, 5 xfailed, 430 warnings in 498.14s (0:08:18) ==============================
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

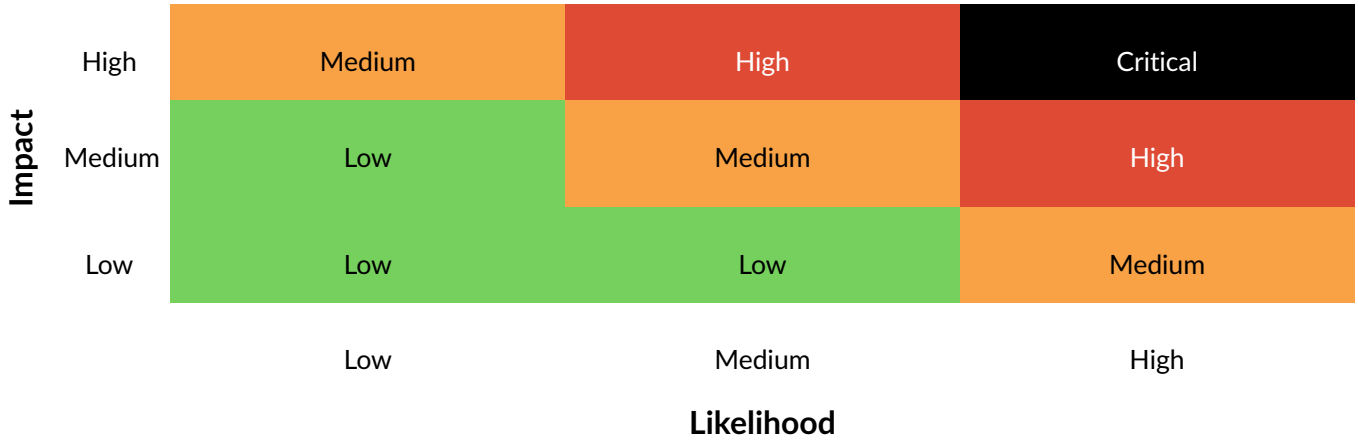

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].