TERM FINANCE

# Term Finance Contracts

## Security Assessment Report

*Version: 3.0*

**April, 2023**

# Contents

# Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Term Finance smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contracts, 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 contracts. 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 and were assessed at commit e57734f. Retesting activities targeted commit e883f0e.

The scope of this assessment included the following contracts:

- `contracts/Authenticator.sol`
- `contracts/TermAuction.sol`
- `contracts/TermAuctionBidLocker.sol`
- `contracts/TermAuctionOfferLocker.sol`
- `contracts/TermController.sol`
- `contracts/TermEventEmitter.sol`
- `contracts/TermInitializer.sol`
- `contracts/TermPriceConsumerV3.sol`
- `contracts/TermRepoCollateralManager.sol`
- `contracts/TermRepoLocker.sol`
- `contracts/TermRepoRolloverManager.sol`
- `contracts/TermRepoServicer.sol`
- `contracts/TermRepoToken.sol`

*Note: the OpenZeppelin libraries and external 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 contracts. 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`

Output for these automated tools is available upon request.

## Findings Summary

The testing team identified a total of 27 issues during this assessment. Categorized by their severity:

- Critical: 10 issues.
- High: 5 issues.
- Low: 5 issues.
- Informational: 7 issues.

## Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Term Finance's smart contracts. 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 |
|---|---|---|---|
| TRM-01 | Bids only transfer collateral for the first token type | Critical | Resolved |
| TRM-02 | Unlocking offers does not return any funds | Critical | Resolved |
| TRM-03 | Cancelling an auction refunds the wrong amount for unrevealed offers | Critical | Resolved |
| TRM-04 | Liquidations fail with "Division by Zero" errors | Critical | Resolved |
| TRM-05 | Rollover bids are set to maximum price, causing reverts | Critical | Resolved |
| TRM-06 | Borrowers cannot withdraw excess collateral after liquidation or default | Critical | Resolved |
| TRM-07 | No checks for non-zero output address when using `ecrecover()` | Critical | Resolved |
| TRM-08 | `authenticate()` tracked nonces not unique per address | Critical | Resolved |
| TRM-09 | If a transaction reverts, its authentication token can be reused by any user to authenticate any transaction | Critical | Resolved |
| TRM-10 | Insufficient Permissions Granted to New Auctions | Critical | Resolved |
| TRM-11 | Authentication tokens could be reused from another chain or project | High | Resolved |
| TRM-12 | `getCollateralBalances()` returns tokens at the zero address if a user does not have a balance in a collateral token | High | Resolved |
| TRM-13 | Bid price not accounted for in `_isInInitialCollateralShortFall()` | High | Resolved |
| TRM-14 | Accounting discrepancy in rollover bids | High | Resolved |
| TRM-15 | Predictable price hashes | High | Resolved |
| TRM-16 | Auction settings can change after an auction starts | Low | Resolved |
| TRM-17 | No protection against initialisation of implementation contracts | Low | Resolved |
| TRM-18 | Use of `transfer()` and `transferFrom()` | Low | Resolved |
| TRM-19 | Minting of tokens does not decrement `mintExposureCap` | Low | Closed |
| TRM-20 | Maximum bid and offer prices are scaled up | Low | Resolved |
| TRM-21 | Excessive gas consumption | Informational | Closed |
| TRM-22 | Protect against risk of front-running `initialize()` functions during protocol deployment | Informational | Resolved |
| TRM-23 | Issues relating to collateral tokens sudden price swings | Informational | Closed |
| TRM-24 | Consider unusual ERC20 token semantics | Informational | Closed |
| TRM-25 | Query about irretrievable tokens | Informational | Closed |

| | | | |
|---|---|---|---|
| TRM-26 | No way to remove from contract registry | Informational | **Resolved** |
| TRM-27 | Miscellaneous general comments | Informational | **Resolved** |

| TRM-01 | Bids only transfer collateral for the first token type | | |
|---|---|---|---|
| Asset | `TermAuctionBidLocker.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

When a bid is submitted, its validity is assessed against all the different collateral tokens in the bid added together. However, only the first collateral token is actually transferred.

A user could submit a large bid with collateral in two tokens: A and B. The amount of token A could be tiny and the amount of token B could be very large. The system would accept the submitted bid and potentially allow this user to borrow a large amount of the purchase token whilst only making the small token A collateral payment.

The function which assesses whether a bid has sufficient collateral is `TermAuctionBidLocker._isInInitialCollateralShortFall()` On line [**434**], this function loops through the array `collateralTokens_`, calculates their cumulative value, and on line [**450**] compares this to the bid's repurchase price.

The code which stores an accepted bid only processes the first collateral token, however. This code is on line [**231**] to line [**272**] and it consistently assesses only `bidSubmission.collateralTokens[0]` and `bidSubmission.collateralAmounts[0]` including, crucially, in the calls to `termRepoCollateralManager.auctionLockCollateral()` where the collateral tokens are transferred.

## Recommendations

If multiple collateral tokens are allowed in a bid, modify the code of `_lock()` to process multiple tokens at each step.

If only one collateral token is to be allowed per bid, review and modify the code of `TermAuctionBidLocker.sol`.

Consider that the system may be significantly simplified by limiting bids to only one kind of collateral token. Multiple collateral bids may be rare, and yet they add significant system complexity which results in higher gas costs and the potential for further security issues. A user could still submit multiple separate bids in the same auction using different collateral tokens.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - multiple collaterals are now correctly supported.

| **TRM-02** | Unlocking offers does not return any funds | |
|---|---|---|
| Asset | `TermAuctionOfferLocker.sol` | |
| Status | **Resolved:** See Resolution | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

When a user unlocks (ie. cancels) an offer, the offer is deleted from the protocol, but the purchase tokens are not returned to the user.

On line [**235**] of `TermAuctionOfferLocker.sol`, a call to `termRepoServicer.unlockOfferAmount()` should return the purchase tokens for an offer to the offerer. However, it is called with the argument `offerToUnlock.amount`. `offerToUnlock` is declared on line [**228**] as a variable of type `storage`. Storage variables are simply pointers, or references to existing storage locations. Therefore, when the variable which `offerToUnlock` points to is deleted on line [**230**], the contents of `offerToUnlock` are deleted too. This means that `offerToUnlock.amount` will be zero, and so no tokens will be returned.

## Recommendations

Use a `memory` variable to copy values that are about to be deleted from storage. The variable type on line [**228**] could be changed to `memory`, but gas could be saved by instead copying only the information required:

```
uint256 amountToUnlock = offers[id].amount;
```

Note that `uint256` variables inside functions are automatically of type `memory`.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - `amountToUnlock` is now set to `offers[id].amount`.

| TRM-03 | Cancelling an auction refunds the wrong amount for unrevealed offers | | |
|---|---|---|---|
| Asset | `TermAuction.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

When the admin cancels an auction, an incorrect amount of purchase tokens is returned for unrevealed offers.

The array `sortedOffers` is used on line [**1162**], while it should be `unrevealedOffers`. As a result, the amount for the revealed offer of the corresponding unrevealed offer index is returned instead.

```
1157 // Return unrevealed offer funds.
     for (i = 0; i < unrevealedOffers.length; i++) {
1159   termAuctionOfferLocker.unlockOfferPartial(
         unrevealedOffers[i].id,
1161     unrevealedOffers[i].offeror,
         sortedOffers[i].amount
1163   );
     }
```

If the value of the index `i` is greater than the maximum index in `sortedOffers`, the transaction would revert. Otherwise, the wrong amount is refunded.

## Recommendations

Change line [**1162**] to `unrevealedOffers[i].amount`.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - `unrevealedOffers` array is now used in `termAuctionOfferLock`
function.

| TRM-04 | Liquidations fail with "Division by Zero" errors | | |
|---|---|---|---|
| Asset | `TermRepoCollateralManager.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

Complete liquidations always fail with a "Division by Zero" error due to the calculations referring to the repaid balance.

At the end of `batchLiquidation()`, a function `_withinNetExposureCapOnLiquidation()` is called, which uses the following in its calculations:

```
uint256 getBorrowerRepurchaseObligation = termRepoServicer.getBorrowerRepurchaseObligation(borrower);

// ...snip...

div_(
  excessEquity,
  termPriceOracle.usdValueOfTokens(
    purchaseToken,
    termRepoServicer.getBorrowerRepurchaseObligation(borrower)
  )
);
```

For a complete liquidation, `getBorrowerRepurchaseObligation()` will always be 0 as, during liquidations, the call to `termRepoServicer.liquidatorCoverExposure()` on line [**339**], will reduce the borrower's repurchase obligation to zero.

The above snippet of code will then always fail with division by `0`, reverting the liquidation transaction and disabling the ability to fully liquidate the account.

## Recommendations

Modify `_withinNetExposureCapOnLiquidation()` to recognise borrower's repurchase obligation being `0`.

This could be achieved by adding the following at the beginning of the `_withinNetExposureCapOnLiquidation()` function:

```
uint256 getBorrowerRepurchaseObligation = termRepoServicer
.getBorrowerRepurchaseObligation(borrower);

if ( getBorrowerRepurchaseObligation == 0 ) { return true; } // <= New code
```

If the borrower has no repurchase obligation, they must be within the levels defined by `netExposureCapOnLiquidation`, and so it is safe to return `true` and avoid both the division by zero and all the unnecessary calculations.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - complete liquidations now do not fail with "Divison by Zero".

| TRM-05 | Rollover bids are set to maximum price, causing reverts | | |
|---|---|---|---|
| Asset | `TermAuction.sol`, `TermAuctionBidLocker.sol`, `TermRepoRolloverManager.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

When processing an auction with at least one rollover bid, the transaction to complete the auction reverts.

In `TermAuctionBidLocker._fillRevealedBidsForAuctionClearing()`, all rollover bids have their price set to `type(uint256).max` on line [**481**]. Rollover bids are submitted with a bid price hash `TermRepoRolloverElectionSubmission.rolloverBidPriceHash` to the function `TermRepoRolloverManager.electRollover()`. Also, all bids provided to `_fillRevealedBidsForAuctionClearing()` are in an array called `revealedBids` of bids with revealed prices. It is unclear why the rollover prices are being overwritten.

When processing auctions containing rollover bids and standard revealed bids, the code reverts with an integer overflow in `TermAuction._calculateClearingPrice()` when the rollover price was added to another price in this section of code:

```
uint256 finalClearingPrice = 0;
if (clearingOffset > state.currentOfferIndex) {
  finalClearingPrice =
    (sortedOffers[0].offerPriceRevealed +
      sortedBids[
        _minUint256(
          sortedBids.length - 1,
          state.currentBidIndex + clearingOffset
        )
      ].bidPriceRevealed) /
    2;
} else {
  // ...
```

## Recommendations

Review the logic of setting rollover bids prices to `type(uint256).max` in `_fillRevealedBidsForAuctionClearing()` and consider a method for processing rollover bids with their submitted prices.

## Resolution

This finding has been addressed during initial testing in PR 681.

| TRM-06 | Borrowers cannot withdraw excess collateral after liquidation or default | | |
|---|---|---|---|
| Asset | `TermRepoCollateralManager.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

Borrowers are unable to withdraw excess collateral after they are entirely liquidated or defaulted. The excess collateral will be locked within the `TermRepoLocker` contract.

This issue stems from the check in `externalUnlockCollateral()` on line [**282**]:

```
281  address borrower = termAuth.user;
     if (termRepoServicer.getBorrowerRepurchaseObligation(borrower) == 0) {
283    revert ZeroBorrowerRepurchaseObligation();  // SigP: This will revert when loan paid in full
     }
285  _unlockCollateral(borrower, collateralToken, amount);
     if (isBorrowerInShortfall(borrower)) {
287    revert CollateralBelowMaintenanceRatios(borrower, collateralToken);
     }
```

`termRepoServicer.getBorrowerRepurchaseObligation(borrower)` will always return `0` if the borrower's owed balance has been fully repaid. In the case of a liquidation or a default, it is possible for this to happen without all of the collateral tokens being paid to the liquidator. The excess collateral tokens would then remain in the `TermRepoLocker` contract, marked as belonging to the borrower. If the borrower attempts to retrieve those tokens, however, the test on line [**282**] will evaluate to `true` and the `revert` on line [**283**] will be triggered.

## Recommendations

Allow borrowers to withdraw leftover collateral after their repurchase obligation has been entirely repaid through liquidation or default.

## Resolution

This finding has been addressed in PR 757.

| TRM-07 | No checks for non-zero output address when using `ecrecover()` | | |
|---|---|---|---|
| Asset | `contracts/Authenticator.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

There are no checks to ensure `ecrecover()` does not return zero.

As such, any invalid signature would be treated as valid when paired with a zero address. The zero address (stored in `termAuth.user`) and an invalid signature can then be used to authenticate transactions.

From `Authenticator` line [**100**]:

```
 82  /// Verifies a signature
     /// @param termAuth The `TermAuth` struct containing user address, nonce, and signature
 84  /// @return bool A boolean testing whether or not a signature is valid
     function authenticate(TermAuth memory termAuth) public returns (bool) {
 86    if (usedNonces[termAuth.nonce]) {
         revert NonceAlreadyUsed(termAuth.nonce);
 88    }
       usedNonces[termAuth.nonce] = true;
 90
       (uint8 v, bytes32 r, bytes32 s) = splitSignature(termAuth.signature);
 92
       string memory header = "\x19Ethereum Signed Message:\n32";
 94    bytes32 check = keccak256(
         abi.encodePacked(
 96        header,
           keccak256(abi.encodePacked(termAuth.nonce, termAuth.user))
 98      )
       );
100    return ecrecover(check, v, r, s) == termAuth.user;      // SigP: should be checking ecrecover is non-zero
     }
```

## Recommendations

Perform checks on the return value from `ecrecover()` and revert if the value is `0`.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - the function now reverts with `InvalidSignature` error if `ecrecover()` returns `0`.

| **TRM-08** | `authenticate()` tracked nonces not unique per address | | |
|---|---|---|---|
| Asset | `Authenticator.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

`usedNonces` tracks all nonces used in authenticated transactions, however, they are not tracked on a per-address basis.

As a result, if a transaction from one wallet address used nonce `X`, any other transaction from a different wallet address with the same nonce `X` will revert. This will cause unexpected and arbitrary denial of service conditions for end users.

## Recommendations

Track all nonces per unique address, not as a collective.

Modify `usedNonces` to be a mapping of an address to an integer (nonce) and use it accordingly.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - `Authenticator` now tracks used nonces on a per user basis.

| TRM-09 | If a transaction reverts, its authentication token can be reused by any user to authenticate any transaction | | |
|---|---|---|---|
| Asset | `Authenticator.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

Authentication tokens can be reused in future transactions when an original transaction reverted.

Each transaction in `Term Finance` requires a special authentication token created by the address which is being affected. This does not have to be the same address as the one submitting the transaction. Once submitted in a completed transaction, these tokens are marked as used and so cannot be reused.

However, reverted transactions cannot modify the state of user accounts on the blockchain. This means that the authentication token will not and cannot be marked as used in a reverted transaction. Another user can then take this token and use it at any time to authenticate any single transaction from that account.

The exploit scenarios for this vulnerability are as wide as anything authorised by `Authenticator.sol` and so could increase in future. Within the scope of the system being reviewed however, one exploit could be to submit a very large, high bid immediately before an auction closes. This would result in a user's funds being transferred from their address and locked in `Term Finance` until the repurchase date. In the worst case scenario, these funds could also be liquidated if their value drops, which is a risk the user did not consent to.

`TermAuth` authentication tokens are validated in `Authenticator.authenticate()`. Each contains a `nonce` value, which is recorded in `usedNonces[termAuth.nonce]` on line [**89**] so that it cannot be reused. This is tested on line [**86**]. However, a reverted transaction would revert all storage changes, including this, and so `usedNonces[termAuth.nonce]` would remain `False` for the token.

## Recommendations

Include a hash of the transaction parameters, the destination contract address, the chain ID, and an expiry time in the `TermAuth` authentication token so that it can only be used to authorise the intended transaction.

Note that, in the case of a reverted transaction, the authorisation token would still be usable before that expiry time by any other user, but only to authorise the original intended transaction.

## Resolution

Based on a retest of commit `e883f0e`, the issue is now resolved - calldata, nonce and expiration timestamp are now signed.

Note, `Authenticator.authenticate()` function takes both `termAuth` and `txMsgData` as parameters.

All checks for expiration timestamp and nonce are performed on values extracted from `termAuth` parameter passed to the function, not the `txMsgData` that is being signed. As a result, when calling `Authenticator.authenticate()`

directly, nonce and expiration timestamp could be modified in `termAuth` parameter to bypass necessary checks and successfully replay previous transactions.

However, this did not appear to be exploitable in the reviewed code base due to the way `Authenticator.authenticate()` is called by the protocol. Every function call requiring authentication contains `termAuth` and invokes `Authenticator.authenticate()` with `msg.data` parameter. As such discrepancy between `termAuth` and `txMsgData` contents cannot occur.

Significant care should be taken to ensure this approach is consistent throughout the protocol and any future developments.

The development team has been made aware of this and acknowledged it.

| TRM-10 | Insufficient Permissions Granted to New Auctions | | |
|---|---|---|---|
| Asset | `TermRepoCollateralManager.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

Newly reopened auctions linked to `TermRepoCollateralManager` do not have necessary `AUCTION_LOCKER` permissions set.

Lack of sufficient permissions set on new auctions will prevent new auctions from ever completing.

## Recommendations

Grant `AUCTION_LOCKER` permissions to a new auction in `TermRepoCollateralManager.reopenToNewAuction()`, such as:

```
function reopenToNewAuction(TermAuctionGroup calldata termAuctionGroup)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
  _grantRole(
      AUCTION_LOCKER,
      address(termAuctionGroup.auction)
  );

//...
```

## Resolution

This finding has been addressed in PR 759.

| TRM-11 | Authentication tokens could be reused from another chain or project | | |
|---|---|---|---|
| Asset | `Authenticator.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: High | Impact: High | Likelihood: Medium |

## Description

Authentication tokens are not unique per chain and could be reused on another chain.

Each transaction in `Term Finance` requires a special authentication token created by the address which is being affected. This does not have to be the same address as the one submitting the transaction. Once submitted in a completed transaction, these tokens are marked as used and so cannot be reused.

However, if `Term Finance` deploys on multiple chains, authentication tokens that are marked as used on one chain would not be marked as used on another. They could then be used by any user to authorise any `Term Finance` transaction on any chains where that token has not already been used.

Even if `Term Finance` never deploys on another chain, it would only require another project to fork the code and that project's tokens would be usable on `Term Finance` and vice versa. This is because the tokens only encode a user's address and a single counter, called the `nonce`.

Reused tokens could authorise any transaction that requires a `TermAuth` authentication token.

## Recommendations

Include a hash of the transaction parameters, the destination contract address (or the address of `Authenticator`), the chain ID, and an expiry time in the `TermAuth` authentication token so that it can only be used to authorise the intended transaction on the intended protocol on the intended blockchain.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - `block.chainid` and transaction's contract address is now included as a part of the hash being signed.

| **TRM-12** | `getCollateralBalances()` returns tokens at the zero address if a user does not have a balance in a collateral token | | |
|---|---|---|---|
| Asset | `TermRepoCollateralManager.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: High | Impact: High | Likelihood: Medium |

## Description

The function `TermRepoCollateralManager.getCollateralBalances()` returns the zero address as the token address if a user does not have a balance in a given token.

This causes rollover bids to fail as `TermAuctionBidLocker._isInInitialCollateralShortFall()` loops through all of the tokens returned and calls `termPriceOracle.usdValueOfTokens()` for each one. This call reverts when called with the zero address as the token's address.

This issue occurs because the return values of `TermRepoCollateralManager.getCollateralBalances()` are set to arrays of length `collateralTokens.length` on line [**543**] but the token addresses are assigned to these arrays conditionally on whether the user has collateral on line [**555**]. Therefore, tokens with no collateral remain as the contents of `memory`.

## Recommendations

Either dynamically set the length of the returned arrays after determining how many token balances will be returned, or else return all token addresses, with zero balances where appropriate, and handle this returned data accordingly.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - `TermAuctionBidLocker._isInInitialCollateralShortFall()` now has a check to omit zero balances of collateral tokens.

| TRM-13 | Bid price not accounted for in `_isInInitialCollateralShortFall()` | | |
|---|---|---|---|
| Asset | `TermAuctionBidLocker.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: High | Impact: Medium | Likelihood: High |

## Description

The calculated repurchase value on line [432] of `TermAuctionBidLocker.sol` does not take into account the submitted bid price. The check is only performed on the bid amount:

```
Exp memory repurchasePriceUSDValue = termPriceOracle.usdValueOfTokens(
   purchaseToken,
   bidAmount
);
```

As a result, a user could submit a large bid without providing sufficient collateral to service the interest on their loan, and thereby win an auction.

Note, this would make them under immediate threat of liquidation after conclusion of the auction, but nonetheless they would be able to participate in and complete the auction successfully.

## Recommendations

When calculating a position's margin in `_isInInitialCollateralShortFall()`, take into account the submitted bid price to ensure the user puts up sufficient collateral to cover the eventual repurchase price.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - `_isInInitialCollateralShortFall()` now takes into account repurchase price.

| TRM-14 | Accounting discrepancy in rollover bids | | |
|---|---|---|---|
| Asset | `TermRepoCollateralManager.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: High | Impact: High | Likelihood: Medium |

## Description

Discrepancy between balances of `lockedCollateralLedger` and `encumberedCollateralBalances` could lead to accounting errors and unexpected reverts.

When a rollover bid is transferred to a new `TermRepoCollateralManager`, `TermAuction` calls `TermRepoCollateralManager.acceptRolloverCollateral()` after transferring the collateral tokens. However, `acceptRolloverCollateral()` only modifies the value of `lockedCollateralLedger[borrower][collateralToken]`. It does not increase `encumberedCollateralBalances[collateralToken]` by the same amount, as it occurs in `_lockCollateral`.

This could lead to an accounting error that could lead to reverts when `_unlockCollateral()` is called for the borrower and `encumberedCollateralBalances[collateralToken]` could be reduced below zero.

## Recommendations

Review the logic of TermRepoCollateralManager.acceptRolloverCollateral().

Modify balances of `encumberedCollateralBalances[collateralToken]` by the same amount as `lockedCollateralLedger[borrower][collateralToken]`

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - `acceptRolloverCollateral` now also increments balances in `encumberedCollateralBalances` ledger.

| **TRM-15** | Predictable price hashes | | |
|---|---|---|---|
| Asset | `TermAuctionBidLocker.sol`, `TermAuctionOfferLocker.sol`, `TermRepoRolloverManager.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: High | Impact: High | Likelihood: Medium |

## Description

Generated price hashes are easy to guess.

Bids and offers send a hash of their price as a mechanism for locking in a price whilst not revealing the price publicly. The method of generating these hashes is predictable, the same price will always result in the same hash. Because of this, it is easy to generate lists of all price hashes and test them against submitted hashes, thereby discovering hidden bid prices.

Similarly, rollover bids are likely to use the same price, and it will be clear if this is the case as the price hash would be unchanged.

A price hash is generated by the simple code:

```
hash = keccak256(abi.encode(price))
```

This code can be reproduced and run quickly to test a large range of potential prices. Once a realistic range has been compiled, a simple script would be able to look up any given price hash in a database of price hashes.

## Recommendations

Consider hashing more information within the price hash. The goal is to add additional levels of entropy, such that the hash produced would not rely on easily predictable factors. It should also change with each bid, so that a single lookup table of prices to their hashes could not be compiled.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - price hashes now also contain a nonce (salt).

| TRM-16 | Auction settings can change after an auction starts | | |
|---|---|---|---|
| Asset | `TermAuction.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Medium |

## Description

The function `TermAuction.pairTermContracts()` is only restricted by role, and can therefore be called after an auction starts.

It could be used to change auction settings after it has already started, potentially causing collateral and loan tokens to be moved to the wrong locker.

## Recommendations

Add a modifier to `TermAuction.pairTermContracts()` that would prevent its use after the auction has started.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - Term contracts can now be initialised only once.

| **TRM-17** | No protection against initialisation of implementation contracts | | |
|---|---|---|---|
| Asset | `contracts/*` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Low | Impact: Medium | Likelihood: Low |

## Description

Majority of reviewed contracts are using a proxy model to allow upgradeability. One known issue of proxy models is that the implementation contracts themselves can sometimes be directly initialised by other users. In many cases, this has little to no impact. However, if a malicious user can find a way to `selfdestruct` the implementation contract, the proxy contract becomes unusable.

Whilst no method of calling `selfdestruct` was found at the time of the review, it is recommended to lock implementation contracts from being initialised by third parties, especially when the contracts' `initialize` functions have no access control, as is the case here.

## Recommendations

For any contract using Openzeppelin's `Initializable` contract, it can be protected from third party initialisation by adding the following code:

```
constructor() {
  _disableInitializers();
}
```

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - `_disableInitializers();` was added to the constructor of implementation contracts.

| TRM-18 | Use of `transfer()` and `transferFrom()` | | |
|---|---|---|---|
| Asset | `TermRepoLocker.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Low | Impact: Medium | Likelihood: Low |

## Description

`TermRepoLocker` uses `transfer()` and `fransferFrom()` functions.

Some ERC20 tokens don't return a value in their `transfer` and `transferFrom()` functions. If there are no checks performed on the returned value (or if there is a value returned), the function may not revert on failed transfers and this may lead to further errors.

`SafeERC20` functions `safeTransfer()` and `safeTransferFrom()` automatically check and assert the boolean return value of a transfer function.

## Recommendations

Use SafeERC20 functions such as `safeTransfer()` and `safeTransferFrom()` to ensure that the return value of the transfer call is checked and handled properly, if there is a return value.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - `safeTransfer()` and `safeTransferFrom()` are now used instead.

| TRM-19 | Minting of tokens does not decrement `mintExposureCap` | | |
|---|---|---|---|
| Asset | `TermRepoToken.sol` | | |
| Status | **Closed:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

The `_mint()` function of `TermRepoToken` does not decrement the `mintExposureCap` counter after minting new tokens.

If not updated, it is possible to mint more tokens than the set cap.

Note, there is a function `decrementMintExposureCap()` that decrements `mintExposureCap` when called manually. It is advised to incorporate it into mint-related functions, instead of calling it manually.

## Recommendations

Decrement `mintExposureCap` after minting new tokens. This can be done by incorporating `decrementMintExposureCap()` into all minting related functions.

## Resolution

The development team has advised:

> "This is not a bug, the `mintExposureCap` is not meant to be decremented when tokens are minted from auction. It is only modified if the mint is thru the function `mintOpenExposure()`."

| TRM-20 | Maximum bid and offer prices are scaled up | | |
|---|---|---|---|
| Asset | `TermAuctionBidLocker.sol`,`TermAuctionOfferLocker.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

`TermAuctionBidLocker.MAX_BID_PRICE()` and `TermAuctionOfferLocker.MAX_OFFER_PRICE()` are both set at `10^22`. As clearing prices are scaled at `10^9`, this value is not the commented 10,000%, but rather `10^13` percent.

## Recommendations

Consider setting `TermAuctionBidLocker.MAX_BID_PRICE()` and `TermAuctionOfferLocker.MAX_OFFER_PRICE()` to `10^13`.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - tender prices are now set to 18 decimals.

| TRM-21 | Excessive gas consumption |
|---|---|
| Asset | `contracts/*` |
| Status | **Closed:** See Resolution |
| Rating | Informational |

## Description

The testing team did note that many processes in the code consume very large quantities of gas. This is not a security issue, but it is nevertheless impactful enough to merit significant examination.

In particular, the logic for completing auctions and calculating the clearing price uses loops within loops. It also involves multiple external calls within loops. These are classic causes of excessive gas usage. Test calls made to `TermAuction.completeAuction()` cost 2.5 million gas with 15 bids and offers and 4 million gas with 25. At 152 bids and offers, the gas usage to call TermAuction.completeAuction() was over 26 million. It is unlikely this will be viable on a public blockchain.

Below are some smaller cases where code can be changed to save gas. This list is by no means exhaustive.

1. `TermAuction._findBidIndexForPrice()` line [**167**] the decrement of `currentBidIndex` could be moved to before line [**162**], thereby removing the need to perform the calculation again in `sortedBids[currentBidIndex - 1]`. A similar logic applies on line [**217**].
2. `TermAuction._findBidIndexForPrice()` the variable `foundBidPrice` is only used in a test to decide whether to modify itself. It can be removed entirely. Also, check that this is not symptomatic of a mistake in the logic of this function. The same is true of `foundOfferPrice` in `TermAuction._findOfferIndexForPrice()`
3. `TermAuction._calculateClearingPrice()` the block starting on line [**364**] uses the expression `_minUint256(state.cumsumBids, state.cumsumOffers)` three times. Repeating calculations almost always costs more gas than storing the result in a memory variable.
4. `TermAuction._findLastIndexForPrice()` line [**515**] the test `sortedOffers.length > 0` is unnecessary as the value `sortedOffers[i].amount` is accessed in the previous line and this would cause a revert if the array had zero length.
5. `TermAuction._assignBids()` line [**786**] the test `(i - innerIndex) != k` will always be true as it takes place within an else block of the test `(i - innerIndex) == k`.
6. `TermAuctionBidLocker.revealBids()` the modifier `onlyWhileAuctionRevealing` will run once for the call to this function, and then again for every iteration of the loop as it calls `this.revealBid()`.
7. `TermRepoRolloverManager.electRollover()` makes the external call `termRepoServicer.getBorrowerRepurchaseObligation(borrower)` twice: on line [**130**] and line [**156**].
8. `TermRepoCollateralManager._withinNetExposureCapOnLiquidation()` the calculation of whether `haircutUSDTotalCollateralValue` is within `netExposureCapOnLiquidation` could be streamlined to a single ratio comparison instead of a subtraction followed by a ratio comparison.
9. `TermRepoCollateralManager._withinNetExposureCapOnLiquidation()` calculates the expression `termPriceOracle.usdValueOfTokens(purchaseToken,getBorrowerRepurchaseObligation)` multiple times.

`TermRepoCollateralManager._withinNetExposureCapOnLiquidation()` stores the return value of `termRepoServicer.getBorrowerRepurchaseObligation(borrower)` in a memory variable and then makes the call again on line [**921**] and line [**931**].

10. `TermRepoCollateralManager._withinNetExposureCapOnLiquidation()` has an expression on line [**917**] which is not being used. These lines could be removed.
11. When incrementing a variable, if the return value is not being used, the format `++i` uses less gas than `i++`.

## Recommendations

For the calculation of the clearing price, consider the following approaches:

1. Fetch the price of each token once during a transaction and reuse this value instead of repeatedly querying oracles.
2. When revealing bid and offer prices, try to calculate, store and update cumulative data that will help facilitate a rapid final calculation.
3. Investigate whether the final algorithm can be restructured entirely to facilitate a faster, simpler calculation. Some mathematical research may be required.
4. Alternatively it may be possible to estimate a starting value that will be generally closer to the final clearing price and iterate from there, thus reducing the number of iterations.

For the smaller issues listed above, review the code and consider changes to reduce unnecessary gas consumption.

## Resolution

The development team has advised:

> "As a result of our Runtime audit, we have totally rewritten the clearing price calculation to reduce gas prices by (around) 50%. We have addressed the other gas optimizations suggested in PR718"

| TRM-22 | Protect against risk of front-running `initialize()` functions during protocol deployment |
|---|---|
| Asset | `contracts/*` |
| Status | **Resolved:** See Resolution |
| Rating | Informational |

## Description

There are no access control checks on `initialize()` functions used to configure the protocol during the deployment.

If there is a period when the contracts have been deployed but not initialised, a malicious attacker could front-run the initialisation process by calling `initialize()` functions to set their own parameters, e.g. add their own addresses as admin or set incorrect protocol values.

The relevant contracts include:

- `contracts/Authenticator.sol`
- `contracts/TermAuction.sol`
- `contracts/TermAuctionBidLocker.sol`
- `contracts/TermAuctionOfferLocker.sol`
- `contracts/TermController.sol`
- `contracts/TermEventEmitter.sol`
- `contracts/TermInitializer.sol`
- `contracts/TermPriceConsumerV3.sol`
- `contracts/TermRepoCollateralManager.sol`
- `contracts/TermRepoLocker.sol`
- `contracts/TermRepoRolloverManager.sol`
- `contracts/TermRepoServicer.sol`
- `contracts/TermRepoToken.sol`

## Recommendations

Depending on the current deployment procedure, it is possible that this issue is already addressed. However, the partial nature of `TermInitializer` gives the testing team some pause. Ideally, consider reworking `TermInitializer` to handle the entire deployment and initialisation process in a single transaction. If this is not desired, ensure that the deployment process uses facilities such as the `Hardhat` deployment functions to initialise all contracts in the same transaction that they are deployed in.

### Resolution

The development team has confirmed that they are using Hardhat plugin to deploy all proxies, which is a preferred approach as it is all done in a single transaction.

| TRM-23 | Issues relating to collateral tokens sudden price swings |
|---|---|
| Asset | `TermRepoCollateralManager.sol` |
| Status | **Closed:** See Resolution |
| Rating | Informational |

## Description

The development team asked the testing team to consider the case of a sudden price drop in the value of one or more collateral tokens.

It is true that, after a significant price drop, calculations in `_collateralSeizureAmounts` will return repayment values larger than all of the available collateral balance put up by the borrower.

From `batchLiquidation()`:

```
(
    collateralSeizureAmount,
    collateralSeizureProtocolShare
) = _collateralSeizureAmounts(
    closureAmounts[i],
    collateralTokens[i]
);

// ...snip...

termRepoLocker.transferTokenToWallet(
    termController.getProtocolReserveAddress(),
    collateralTokens[i],
    collateralSeizureProtocolShare
);

termRepoLocker.transferTokenToWallet(
    termAuth.user,
    collateralTokens[i],
    collateralSeizureAmount - collateralSeizureProtocolShare    // SigP: This value will be larger than available collateral balance
);
```

In such a case, it will be impossible to liquidate some borrowers until the collateral price returns to levels where the available balance is sufficient to pay out the liquidator and cover protocol fees.

## Recommendations

This is a known issue with collateralised lending protocols. The main mitigation measures are economic management of the protocol's settings. In this case, the values of `maintenanceCollateralRatios`, `initialCollateralRatios` and `liquidatedDamages` in `TermRepoCollateralManager` will control the margins at which liquidations are possible and the rewards for the liquidators. It is necessary to manage these values to mitigate the risk, but risk from a very sudden, rapid price drop will remain. For this reason, choice of collateral is also important: collateral tokens likely to drop in price very suddenly are best avoided.

### Resolution

The devlopment team has acknowledged this issue and no further action has been taken as of 31/03/2023.

| TRM-24 | Consider unusual ERC20 token semantics |
|---|---|
| Asset | `contracts/*` |
| Status | **Closed:** See Resolution |
| Rating | Informational |

## Description

Whilst being compliant with the ERC20 standard, some implementations extend the specification by adding functionality or alternative behaviours to various operations. Some particularly notable examples of this are:

- **Transfer fees -** Where fees are deducted on calls to `transfer` or `transferFrom` and forwarded to either the protocol developers, a treasury, or some other party.
- **Rebasing supply -** Where the total supply of the token changes either on transfer events or due to some other economic policy (e.g., Ampleforth).
- **Interest accrual -** Where holders of a token may accrue interest or other rewards for holding or staking a token.
- **Extreme precision -** Where a token may have abnormally low or high `decimals` (e.g., 0 or 256).

## Recommendations

Consider possible edge cases in ERC20 implementations that exist in-the-wild and adjust design assumptions around these accordingly.

Ensure that these assumptions are enforced by the codebase (where possible) and are explicitly documented. Add dedicated token integration tests to the existing testing suite.

The devlopment team has acknowledged this issue and no further action has been taken as of 31/03/2023.

## Resolution

The devlopment team has acknowledged this issue and no further action has been taken as of 14/04/2023.

| TRM-25 | Query about irretrievable tokens |
|---|---|
| Asset | `TermRepoLocker.sol` |
| Status | **Closed:** See Resolution |
| Rating | Informational |

## Description

The development team asked the testing team to consider the case where gas limits could lock tokens in `TermRepoLocker`.

It is the opinion of the testing team that, with the current logic, gas problems with `completeAuction()` would have rendered the protocol unusable long before the number of bids would reach the levels necessary to cause `TermAuction.cancelAuction()` to fail due to gas limits.

It is conceivable, however, that an auction could become flooded with bids and offers as part of an attack.

In this case, legitimate user funds could always be rescued from `TermRepoLocker` by upgrading it to a new version with a function that allows the contract to transfer out tokens.

## Recommendations

No action required.

## Resolution

The devlopment team has acknowledged this issue and is considering a solution of setting a maximum number of bids/offers allowed per auction, but no further action has been taken as of 14/04/2023.

| TRM-26 | No way to remove from contract registry |
|---|---|
| Asset | `TermController.sol` |
| Status | **Resolved:** See Resolution |
| Rating | Informational |

## Description

There is no way to remove addresses from the registry `termAddresses` maintained by `TermController`.

In the situation where a registered contract is discovered to have a serious vulnerability, it would be desirable to not have it registered as a legitimate contract.

## Recommendations

Consider adding a function to `TermController` that removes entries from `termAddresses`.

## Resolution

Based on a retest of commit `e883f0e`, the issue has been resolved - `unmarkTermDeployed()` function to remove a contract from `TermController` has now been added.

| TRM-27 | Miscellaneous general comments |
|---|---|
| Asset | `contracts/*` |
| Status | **Resolved:** See Resolution |
| Rating | Informational |

## Description

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

1. **Missing NatSpec documentation comments** Many functions throughout the codebase lack either full or any documentation comments at all. This worsens code readability and complicates the use of automated documentation tooling.

   Note that, if a function inherits its documentation from another (i.e., when overriding from an interface), the `@inheritdoc` NatSpec directive can be used to communicate this.

2. **Magic numbers and strings** Avoid magic numbers and strings in favour of explicit constants. The latter improves semantic clarity.

   Some individual examples are,

   - Expected signature length in `Authenticator.sol`
   - Various numeric literals in `dayCountMantissa` calculations
   - EIP712 header in `Authenticator::authenticate`

3. **Debugging artefacts** In `TermAuction.sol` on line 18, Hardhat's `console.log` implementation is left imported

4. **TermInitializer does not need to be upgradeable** The proxy model is probably undesirable for the initialiser contract. It is highly improbable that the contract would be upgraded with its current storage and far more likely that a new version would simply be redeployed.

5. **Test may not achieve anything** The check on line [**290**] of `TermRepoRolloverManager` calls a function `auctionBidLocker.isAuctionBidLocker()`. Most incorrect contracts will not have this function and so will simply revert, not revert with the custom error on line [**291**]. Some contracts may have a `fallback` function which could return `True`. The custom error will only be used if the function (or a fallback) does exist and returns `False`.

6. **Function in interface is not implemented** `ITermAuctionBidLocker.sol` contains the function `collateralTokens(IERC20Upgradeable token)` on line [**21**] but this function is not implemented in the contract.

7. **TermRepoCollateralManager.initialize() allows inappropriate zero values** `initialize()` allows `collateralTokens_[i].liquidatedDamage` to be zero, but this is what is tested to ensure a token is valid collateral in `_isAcceptedCollateralToken()`. It may help avoid some potential configuration problems if a zero value were disallowed for `collateralTokens_[i].liquidatedDamage`.

8. **Zero address checks in initialisers** Consider checking that the supplied arguments are not `address(0)` for the `initialize()` functions in the following instances:

   - `purchaseToken_` in `TermRepoServicer.sol`
   - `purchaseToken_` in `TermRepoCollateralManager.sol`

- `treasuryWallet_` in `TermController.sol`
- `protocolReserveWallet_` in `TermController.sol`

9. **Particular comment issues**
   - `TermAuction.sol` the comment for the functions `_findBidIndexForPrice()` and `_findOfferIndexForPrice()` don't explain what the second return value does (it seems to be a flag for whether a higher or equal price was found).
   - `TermAuction.sol` the comment on line [**354**] mentions offers "lower than" but the function called, `_cumsumOfferAmount()` tests for offers lower or equal to. This could be misleading given the comment on line [**360**] which does say, "higher than or equal to".
   - `TermAuction.sol` the comment for the functions `_findBidIndexForPrice()` and `_findOfferIndexForPrice()` don't explain what the second return value does (it seems to be a flag for whether a higher or equal price was found).
   - The comment on line [**693**] of `TermRepoCollateralManager.sol` is wrong. The parameter is actually a struct of auction contracts.
   - In `TermRepoRolloverManager.electRollover()`, it is not clear from comments that `termRepoRolloverElectionSubmission.rolloverAuction` is an instance of `TermAuctionBidLocker` and not `TermAuction`.
10. **Event emitting issues** It was noted that bid locking events are not being emitted in `TermAuctionBidLocker.lockBids()`. Review and ensure all key events are emitted as required.

## Recommendations

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

## Resolution

The development team have acknowledged these findings, addressing them where feasible, at their discretion.

# 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.

```
tests/test_Authenticator.py xx.                                                                                [  3%]
tests/test_TermAuction.py .x..                                                                                 [  7%]
tests/test_TermAuctionBidLocker.py ...................                                                         [ 28%]
tests/test_TermAuctionOfferLocker.py .........xx.....                                                          [ 45%]
tests/test_TermController.py ...                                                                               [ 48%]
tests/test_TermEventEmitter.py .                                                                               [ 50%]
tests/test_TermInitializer.py ...                                                                              [ 53%]
tests/test_TermPriceConsumerV3.py .                                                                            [ 54%]
tests/test_TermRepoCollateralManager.py ...xxx.xX                                                              [ 64%]
tests/test_TermRepoLocker.py ...                                                                               [ 67%]
tests/test_TermRepoRolloverManager.py ..........                                                               [ 78%]
tests/test_TermRepoServicer.py .......                                                                         [ 85%]
tests/test_TermRepoToken.py .........                                                                          [ 95%]
tests/test_poc_max_price.py .                                                                                  [ 96%]
tests/test_poc_multicollateral.py .                                                                            [ 97%]
tests/test_poc_revert_auth.py .                                                                                [ 98%]
tests/test_poc_rollover_max.py x                                                                               [100%]
```

## 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.

| | | | | |
|---|---|---|---|---|
| Impact | High | Medium | High | Critical |
| | Medium | Low | Medium | High |
| | Low | Low | Low | Medium |
| | | Low | Medium | High |
| | | | Likelihood | |

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].