SHERLOCK

Security Review For Teller

Public Contest Prepared For: **Teller**
Lead Security Expert: **hash**
Date Audited: **December 4 - December 10, 2024**

Introduction

Teller is an extensible lending protocol for OTC loans. Lender groups is a contract stack on top that enables pool-style lending using the OTC loan backend, making for a unique permissionless architecture. (Can lend assets even if not allowlisted by our protocol)

Scope

Repository: teller-protocol/teller-protocol-v2-audit-2024

Branch: merge-train-r5

Audited Commit: d829864a1ddf408218692449b228af36c23001ef

Final Commit: 53c51f9b2bda02b90487630dc4e62d1264bdcf1f

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
3	8

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

hash
OpaBatyo
KupiaSec

0xeix
PeterSR
0xpetern

Flare
mgf15

Issue H-1: Lender group members can be prevented from burning their shares forever

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/24

Found by

KupiaSec, OpaBatyo

Summary

Summary

Adversaries can constantly reset the withdrawal delay of lender group members by performing 0-value `transferFrom` transactions to invoke the `afterTokenTransfer` hook.

Description

Currently there is a delay on withdrawals to prevent sandwich attacks in lender group contracts. Members must invoke `prepareSharesForBurn` by stating how many shares they want to burn and start an internal countdown. Afterwards, members invoke `burnSharesToWithdrawEarnings` which checks whether the delay passed in `burn`

```
 function burn(address _burner, uint256 _amount, uint256 withdrawDelayTimeSeconds)
 ↪  external onlyOwner {

     //require prepared
     require(poolSharesPreparedToWithdrawForLender[_burner] >= _amount,"Shares not
 ↪  prepared for withdraw");
@>   require(poolSharesPreparedTimestamp[_burner] <= block.timestamp -
 ↪  withdrawDelayTimeSeconds,"Shares not prepared for withdraw");


     //reset prepared
     poolSharesPreparedToWithdrawForLender[_burner] = 0;
     poolSharesPreparedTimestamp[_burner] =  block.timestamp;



     _burn(_burner, _amount);
 }
```

This countdown is reset every time a member invokes a share transfer through the `_afterTokenTransfer` hook presumably to prevent users preparing shares in advance by transferring it between one another.

```
 function _afterTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal override {

     //reset prepared
    poolSharesPreparedToWithdrawForLender[from] = 0;
    poolSharesPreparedTimestamp[from] =  block.timestamp;


}
```

Adversaries can perform 0-value `transferFrom` transactions which will always pass as there are no 0-value checks in OZ's version 4.8 `ERC20.sol` used by the protocol. Users will have their countdown constantly reset thus being prevented from withdrawing forever or until a bribe is paid.

Root Cause

- OpenZeppelin's `ERC20.transferFrom` has no 0-value input validation
- `LenderCommitmentGroupShares._afterTokenTransfer` does not perform 0-value input either.

Internal pre-conditions

Group members must have invoked `prepareSharesForBurn`

External pre-conditions

None

Attack Path

1. Group member invokes `prepareSharesForBurn` starting the countdown
2. Adversary invokes `transferFrom(victim,to,0)` minutes before the cooldown expires
3. Cooldown and shares are reset because `_afterTokenTransfer` was triggered
4. Group member is forced to re-prepare their shares
5. Attacker repeats this continuously or until a bribe is paid

Impact

Lender commit group members will have their funds permanently locked in the contract

PoC

No response

Mitigation

Rewrite the hook to be skipped in case of `amount=0`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/76

Issue H-2: Malicious lender can prevent borrower from repayment due to try/catch block revert

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/39

Found by

OpaBatyo, hash

Summary

Insufficient validation for `try/catch` address will disallow borrower's from repaying their loans

Root Cause

A malicious lender can bypass the `try/catch` block covering the repayLoanCallback external call by selfdestructing `loanRepaymentListener`

```
if (loanRepaymentListener != address(0)) {
    require(gasleft() >= 80000, "NR gas");  //fixes the 63/64 remaining issue
    try
        ILoanRepaymentListener(loanRepaymentListener).repayLoanCallback{
            gas: 80000
        }( //limit gas costs to prevent lender preventing repayments
            _bidId,
            _msgSenderForMarket(bid.marketplaceId),
            _payment.principal,
            _payment.interest
        )
```

The `try/catch` block will revert if the call is made to a non-contract address. To avoid this a check for `codesize>0` is kept inside the `setRepaymentListenerForBid` function. But this can be bypassed by the lender `selfdestructing` the `_listener` in the same transaction which will delete the contract

https://github.com/sherlock-audit/2024-11-teller-finance-update/blob/0c8535728f97d37a4052d2a25909d28db886a422/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L1287-L1301

```
function setRepaymentListenerForBid(uint256 _bidId, address _listener) external {
    uint256 codeSize;
```

```
    assembly {
        codeSize := extcodesize(_listener)
    }
    require(codeSize > 0, "Not a contract");
    address sender = _msgSenderForMarket(bids[_bidId].marketplaceId);


    require(
        sender == getLoanLender(_bidId),
        "Not lender"
    );


    repaymentListenerForBid[_bidId] = _listener;
}
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. Lender creates a contract which can `selfdestruct` itself
2. Lender sets this address as the repaymentListener
3. In the same tx, the lender destroys the contract
4. Now the borrower cannot repay because the try/catch block will always revert

Impact

Borrowers will not be able to repay the loan allowing the lender to steal the collateral after the loan will default

PoC

No response

Mitigation

Use .call instead of the try/catch

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/79

Issue H-3: Using original principal amount as due amount inside `liquidateDefaultedLoanWithIncentive` breaks contract accounting leading to lost assets/broken functionalities

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/43

Found by

0xeix, hash

Summary

Using original principal amount as due amount inside `liquidateDefaultedLoanWithIncentive` breaks contract accounting leading to lost assets/broken functionalities

Root Cause

In `liquidateDefaultedLoanWithIncentive`, the amount due is taken as the original principal amount of the bid rather than the remaining to be repaid principal which is incorrect as part of this principal could have already been paid back

```
function liquidateDefaultedLoanWithIncentive(
    uint256 _bidId,
    int256 _tokenAmountDifference
) external whenForwarderNotPaused whenNotPaused bidIsActiveForGroup(_bidId)
 ↪  nonReentrant onlyOracleApprovedAllowEOA {

    //use original principal amount as amountDue

    uint256 amountDue = _getAmountOwedForBid(_bidId);
```

```
function _getAmountOwedForBid(uint256 _bidId )
    internal
    view
    virtual
    returns (uint256 amountDue)
{
    // @audit this is the entire principal amount which is incorrect
```

```
    (,,,, amountDue, , ,  )
     = ITellerV2(TELLER_V2).getLoanSummary(_bidId);
}
```

Parts of the original principal could have been repaid and accounted via this function leading to double counting of principal in `totalPrincipalTokensRepaid`

```
function repayLoanCallback(
    uint256 _bidId,
    address repayer,
    uint256 principalAmount,
    uint256 interestAmount
) external onlyTellerV2 whenForwarderNotPaused whenNotPaused
 ↪  bidIsActiveForGroup(_bidId) {
    totalPrincipalTokensRepaid += principalAmount;
    totalInterestCollected += interestAmount;
```

This leads to several problems:

1. Underflow in `totalPrincipalTokensLended-totalPrincipalTokensRepaid` as `totalPrincipalTokensRepaid` can double count repaid tokens causing bids to revert
2. Lost assets due to `tokenDifferenceFromLiquidations` calculating difference from `totalPrincipal` without considering the repaid assets

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. A loan is created with principal amount == 100 tokenBalanceOfContract == 0

totalPrincipalTokensCommitted == 100 totalPrincipalTokensWithdrawn == 0
totalPrincipalTokensLended == 100 totalPrincipalTokensRepaid == 0
tokenDifferenceFromLiquidations == 0

2. Repayment of 80 principal occurs before the loan gets defaulted tokenBalanceOfContract == 80

totalPrincipalTokensCommitted == 100 totalPrincipalTokensWithdrawn == 0
totalPrincipalTokensLended == 100 totalPrincipalTokensRepaid == 80
tokenDifferenceFromLiquidations == 0

3. Loan defaults and auction settles at price 50 (similarly problematic paths are lenders withdrawing 80 first or the auction settling at higher prices) tokenBalanceOfContract == 80 + 50 == 130

totalPrincipalTokensCommitted == 100 totalPrincipalTokensWithdrawn == 0 totalPrincipalTokensLended == 100 totalPrincipalTokensRepaid == 80 + 100 == 180 => incorrect tokenDifferenceFromLiquidations == (100 - 50 == -50) => incorrect

Now:

- available amount to withdraw will be calculated as (totalPrincipalTokensCommitted + tokenDifferenceFromLiquidations == 50) while there is actually 130 amount of assets available to withdraw causing loss for lenders
- getPrincipalAmountAvailableToBorrow will underflow because (totalPrincipalTokensLended - totalPrincipalTokensRepaid == -80) and no new bids can be accepted

There are more scenarios that arise from the same root cause such as estimated value becoming 0 incorrectly, which will cause division by 0 and hence revert on withdrawals, deposits will be lost or 0 shares will be minted etc.

Impact

Lost assets for users, broken functionalities

PoC

No response

Mitigation

Instead of the totalPrincipal consider the remaining principal ie. `totalPrincipal-repaidPrincipal`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/85

Issue M-1: ERC20.approve Used Instead of Safe Approvals, Causing Pool Failures with Some ERC20s

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/29

Found by

0xpetern, Flare, hash, mgf15

Summary

The function acceptFundsForAcceptBid handles accepting a loan bid, transferring funds to the borrower, and securing collateral. It approves the TELLER_V2 contract to transfer the loan's principal amount using "ERC20.approve".

However, some ERC20s on some chains don't return a value. The most popular example is USDT and USDC on the main net, and as the docs mention it should be compatible on any EVM chain and will support USDT:

> Q: On what chains are the smart contracts going to be deployed? Ethereum Mainnet, Polygon PoS, Arbitrum One, Base
>
> Q: If you are integrating tokens, are you allowing only whitelisted tokens to work with the codebase or any complying with the standard? Are they assumed to have certain properties, e.g. be non-reentrant? Are there any types of [weird tokens](https://github.com/d-xo/weird-erc20) you want to integrate? We absolutely do want to fully support all mainstream tokens like USDC, USDT, WETH, MOG, PEPE, etc. and we already took special consideration to make sure USDT works with our contracts.

Despite this claim, the current implementation of the approve function does not account for tokens like USDT, contradicting the protocol's intentions. Therefore acceptFundsForAcceptBid will never work on the EVM Chain or other related chain and tokens.

Root Cause

In LenderCommitmentGroup_Smart.soL: 556, approve function is used instead of safeApprove. USDT on the main net doesn't return a value, https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7#code. This includes USDC which should work in the protocol. This behavior causes the approve function to revert when interacting with these tokens.

```
function acceptFundsForAcceptBid(
        address _borrower,
        uint256 _bidId,
        uint256 _principalAmount,
        uint256 _collateralAmount,
        address _collateralTokenAddress,
        uint256 _collateralTokenId,
        uint32 _loanDuration,
        uint16 _interestRate
    ) external onlySmartCommitmentForwarder whenForwarderNotPaused whenNotPaused {

        require(
            _collateralTokenAddress == address(collateralToken),
            "Mismatching collateral token"
        );
        //the interest rate must be at least as high has the commitment demands.
↪  The borrower can use a higher interest rate although that would not be
↪  beneficial to the borrower.
        require(_interestRate >= getMinInterestRate(_principalAmount), "Invalid
↪  interest rate");
        //the loan duration must be less than the commitment max loan duration. The
↪  lender who made the commitment expects the money to be returned before this
↪  window.
        require(_loanDuration <= maxLoanDuration, "Invalid loan max duration");

        require(
            getPrincipalAmountAvailableToBorrow() >= _principalAmount,
            "Invalid loan max principal"
        );


        uint256 requiredCollateral = calculateCollateralRequiredToBorrowPrincipal(
            _principalAmount
        );



        require(
             _collateralAmount   >=
                requiredCollateral,
            "Insufficient Borrower Collateral"
        );

        principalToken.approve(address(TELLER_V2), _principalAmount);
//do not have to override msg.sender as this contract is the lender !
        _acceptBidWithRepaymentListener(_bidId);

        totalPrincipalTokensLended += _principalAmount;
```

```
        activeBids[_bidId] = true; //bool for now


        emit BorrowerAcceptedFunds(
            _borrower,
            _bidId,
            _principalAmount,
            _collateralAmount,
            _loanDuration,
            _interestRate
         );
    }
```

The specific part of the function is highlighted her;

```
principalToken.approve(address(TELLER_V2), _principalAmount);

//do not have to override msg.sender as this contract is the lender !
```

https://github.com/sherlock-audit/2024-11-teller-finance-update/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L556

Internal pre-conditions

1. The protocol uses approve calls in functions like acceptFundsForAcceptBid.
2. There is no consideration or error handling for approve calls when interacting with USDT, USDC.

External pre-conditions

1. The protocol operates on EVM-compatible chains where these tokens are prevalent.

Attack Path

1. Users or lenders supply USDT or USDC as the principalToken.
2. Transactions that include approve calls revert, causing the protocol to fail in providing the intended functionality.

Impact

1. The protocol becomes unusable with ERC20 tokens like USDT AND USDC, a widely used stablecoin.
2. Breaks the intended functionality of the protocol.
3. This breaks a criticial function in the protocol and causes the pool to fail.

PoC

No response

Mitigation

Use `safeApprove` instead of `approve`

```
require(
             _collateralAmount   >=
                requiredCollateral,
            "Insufficient Borrower Collateral"
        );
principalToken.safeApprove(address(TELLER_V2), _principalAmount);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/78

Issue M-2: Users can lower the interest rate by dividing a loan into multiple smaller loans

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/34

The protocol has acknowledged this issue.

Found by

KupiaSec

Summary

The `LenderCommitmentGroup_Smart.getMinInterestRate()` function calculates the minimum APR for borrowing. However, this APR is determined based on the utilization ratio, which includes the newly borrowed amount. This allows borrowers to reduce the APR by dividing a loan into multiple smaller loans.

Root Cause

The `LenderCommitmentGroup_Smart.getMinInterestRate()` function calculates the minimum APR for borrowing.
https://github.com/sherlock-audit/2024-11-teller-finance-update/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L1017-1023

```
  function getMinInterestRate(uint256 amountDelta) public view returns (uint16) {
        return interestRateLowerBound +
        uint16( uint256(interestRateUpperBound-interestRateLowerBound)
@>      .percent(getPoolUtilizationRatio(amountDelta )

        ) );
    }
```

However, the APR is calculated based on the utilization ratio, which includes the newly borrowed amount. This allows borrowers to reduce the APR by dividing a loan into multiple smaller loans, creating an unfair advantage for them over other users.
https://github.com/sherlock-audit/2024-11-teller-finance-update/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L1002-1015

```
    function getPoolUtilizationRatio(uint256 activeLoansAmountDelta ) public view
↪   returns (uint16) {

        if (getPoolTotalEstimatedValue() == 0) {
            return 0;
        }

        return uint16(  Math.min(
                            MathUpgradeable.mulDiv(
@>                              (getTotalPrincipalTokensOutstandingInActiveLoans()
↪   + activeLoansAmountDelta),
                                10000  ,
                                getPoolTotalEstimatedValue() ) ,
                        10000  ));

    }
```

Internal pre-conditions

interestRateLowerBound = 0 interestRateLowerBound = 800 // 8% current total estimated value = $20000 current borrowed value = $5000

External pre-conditions

none

Attack Path

If Alice borrows $10000 USD at once, the APR is (5000 + 10000) / 20000 * 8% = 6%

However, Alice borrow $10000 as follows.

1. Alice is going to borrow $10000.
2. Alice borrows $1,000 repeatedly for a total of 10 times.

Then the APR for each 1000 USD is: (5000 + 1000) / 20000 * 8% = 2.4% (6000 + 1000) / 20000 * 8% = 2.8% (7000 + 1000) / 20000 * 8% = 3.2% (8000 + 1000) / 20000 * 8% = 3.6% (9000 + 1000) / 20000 * 8% = 4.0% (10000 + 1000) / 20000 * 8% = 4.4% (11000 + 1000) / 20000 * 8% = 4.8% (12000 + 1000) / 20000 * 8% = 5.2% (13000 + 1000) / 20000 * 8% = 5.6% (14000 + 1000) / 20000 * 8% = 6%

(2.4 + 2.8 + ... + 6) / 10 = 4.2

Therefore, the borrower will only pay an APR of 4.2% on the $10,000 loan. As a result, Alice lowers the interest rate from 6% to 4.2% by splitting a $10,000 loan into ten separate loans of $1,000 each.

Impact

Users can lower the interest rate by dividing a loan into multiple smaller loans.

PoC

none

Mitigation

Middle value should be used instead of end value.

```
    function getPoolUtilizationRatio(uint256 activeLoansAmountDelta ) public view
 ↪  returns (uint16) {

        if (getPoolTotalEstimatedValue() == 0) {
            return 0;
        }

        return uint16(  Math.min(
                            MathUpgradeable.mulDiv(
-                               (getTotalPrincipalTokensOutstandingInActiveLoans()
 ↪  + activeLoansAmountDelta),
+                               (getTotalPrincipalTokensOutstandingInActiveLoans()
 ↪  + (activeLoansAmountDelta + 1) / 2),
                                10000  ,
                                getPoolTotalEstimatedValue() ) ,
                        10000  ));

    }
```

Issue M-3: Attacker can revoke any user from a market

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/37

Found by

PeterSR, hash

Summary

Lack of access control in `revokeLender` allows an attacker to revoke any participant from a market

Root Cause

The delegation version of the `revokeLender` function fails to perform any access control checks allowing any user to revoke any user

```
function _revokeStakeholderViaDelegation(
    uint256 _marketId,
    address _stakeholderAddress,
    bool _isLender,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) internal {
    bytes32 uuid = _revokeStakeholderVerification(
        _marketId,
        _stakeholderAddress,
        _isLender
    );
    // NOTE: Disabling the call to revoke the attestation on EAS contracts
    //        address attestor = markets[_marketId].owner;
    //        tellerAS.revokeByDelegation(uuid, attestor, _v, _r, _s);
}
```

Internal pre-conditions

Attestation should be enabled to observe the impact

External pre-conditions

No response

Attack Path

1. Attacker calls `revokeLender` by passing in any address they wish to revoke from the market

Impact

Attacker can revoke any address they wish from any market making the market unuseable

PoC

No response

Mitigation

Perform access control checks

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/77

Issue M-4: Not updating state before making custom external call can cause borrower's to loose assets due to re-entrancy

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/42

Found by

hash

Summary

Not updating state before making custom external call can cause borrower's to loose assets due to re-entrancy

Root Cause

The details of the repayment is updated only after the external call to the `loanRepaymentListener` is made

https://github.com/sherlock-audit/2024-11-teller-finance-update/blob/0c8535728f97d37a4052d2a25909d28db886a422/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L865-L870

```
function _repayLoan(
    uint256 _bidId,
    Payment memory _payment,
    uint256 _owedAmount,
    bool _shouldWithdrawCollateral
) internal virtual {

    ....
    // @audit attacker can re-enter here. the repayment details are not yet updated
    _sendOrEscrowFunds(_bidId, _payment); //send or escrow the funds

    // update our mappings
    bid.loanDetails.totalRepaid.principal += _payment.principal;
    bid.loanDetails.totalRepaid.interest += _payment.interest;
    bid.loanDetails.lastRepaidTimestamp = uint32(block.timestamp);
```

```
function _sendOrEscrowFunds(uint256 _bidId, Payment memory _payment)
    internal virtual
```

```
{

    ....

    address loanRepaymentListener = repaymentListenerForBid[_bidId];

    // @audit re-enter in this call
    if (loanRepaymentListener != address(0)) {
        require(gasleft() >= 80000, "NR gas");  //fixes the 63/64 remaining issue
        try
            ILoanRepaymentListener(loanRepaymentListener).repayLoanCallback{
                gas: 80000
            }( //limit gas costs to prevent lender preventing repayments
                _bidId,
                _msgSenderForMarket(bid.marketplaceId),
                _payment.principal,
                _payment.interest
            )
        {} catch {}
```

This allows a malicious lender to reenter the `TellerV2` contract and invoke `lenderCloseLoan` seizing the collateral of the borrower as well if the loan is currently defaulted

Internal pre-conditions

1. The repayment should be made after defaultTimestamp has passed

External pre-conditions

No response

Attack Path

1. Defaulting timestmap of loan has passed
2. Borrower does a repayment of 100 which is transferred to the lender. Following this . `repayLoanCallback` is called
3. Lender reenters via the `loanRepaymentListener` and invokes the `lenderCloseLoan` function further seizing the collateral of the borrower
4. Borrower looses both the repayment amount and the collateral

Impact

Borrower will loose repayment amount and also the collateral

PoC

No response

Mitigation

Update the state before the `loanRepaymentListener` call is made

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/81

Issue M-5: Repayer can brick lending functionality of `LenderCommitmentGroup_Smart` by repaying excess

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/46

Found by

OpaBatyo, hash

Summary

`LenderCommitmentGroup_Smart` doesn't handle excess repayments making it possible to brick the lending functionality

Root Cause

The _function performs `totalPrincipalTokensLended-totalPrincipalTokensRepaid` without handling the underflow scenario. This is problematic as excess amount can be repaid for loans which will cause an underflow here

```
function getTotalPrincipalTokensOutstandingInActiveLoans()
    public
    view
    returns (uint256)
{
    return totalPrincipalTokensLended - totalPrincipalTokensRepaid;
}
```

on excess repayments, `totalPrincipalTokensRepaid` will become greater than `totalPrincipalTokensLended`

```
function repayLoanCallback(
    uint256 _bidId,
    address repayer,
    uint256 principalAmount,
    uint256 interestAmount
) external onlyTellerV2 whenForwarderNotPaused whenNotPaused
 ↪  bidIsActiveForGroup(_bidId) {
    totalPrincipalTokensRepaid += principalAmount;
    totalInterestCollected += interestAmount;
```

function in TellerV2.sol contract to repay excess amount. User can specify any amount greater than minimum amount

```
function repayLoan(uint256 _bidId, uint256 _amount)
    external
    acceptedLoan(_bidId, "rl")
{
    _repayLoanAtleastMinimum(_bidId, _amount, true);
}
```

The function `getTotalPrincipalTokensOutstandingInActiveLoans` is invoked before every lending. Hence an underflow here will cause the lending functionality to revert

Another quirk regarding excess repayments is that the lenders of the pool won't obtain the excess repaid amount since it is not accounted anywhere. But this cannot be considered an issue since the lenders are only guarenteed the original lending amount + interest

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. Attacker borrows 100 from the lender pool totalPrincipalTokensLended == 100 totalPrincipalTokensRepaid == 0
2. Attacker repays 101 totalPrincipalTokensLended == 100 totalPrincipalTokensRepaid == 101

Now `getTotalPrincipalTokensOutstandingInActiveLoans` will always revert

Impact

Bricked lending functionality

PoC

No response

Mitigation

In case repaid principal is more, return 0 instead

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/80

Issue M-6: Tokens that revert of zero value transfers can cause reverts on liquidation

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/51

Found by

hash

Summary

Tokens that revert of zero value transfers can cause reverts on liquidation

Root Cause

In the readme the team has mentioned that they would like to know if any wierd token breaks their contract pools

In multiple places token amount which can become zero is transferred without checking the value is zero. This will cause these transactions to revert https://github.com/sherlock-audit/2024-11-teller-finance-update/blob/0c8535728f97d37a4052d2a25909d28db886a422/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L699-L727

```
    IERC20(principalToken).safeTransferFrom(
        msg.sender,
        address(this),
        amountDue + tokensToTakeFromSender - liquidationProtocolFee
    );

    address protocolFeeRecipient =
↪   ITellerV2(address(TELLER_V2)).getProtocolFeeRecipient();


      IERC20(principalToken).safeTransferFrom(
        msg.sender,
        address(protocolFeeRecipient),
         liquidationProtocolFee
    );


    totalPrincipalTokensRepaid += amountDue;
```

```
    tokenDifferenceFromLiquidations += int256(tokensToTakeFromSender -
↪   liquidationProtocolFee );

} else {

    uint256 tokensToGiveToSender = abs(minAmountDifference);



    IERC20(principalToken).safeTransferFrom(
        msg.sender,
        address(this),
        amountDue - tokensToGiveToSender
    );
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

In case liquidation reverts (due to tokensToGiveToSender == -amountDue), the `tokenDifferenceFromLiquidations` won't be updated which will cause the value of the shares to be incorrectly high (because in reality the auction is settling at 0 price)

PoC

No response

Mitigation

Check if amount is non-zero before transferring

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/82

Issue M-7: The `totalPrincipalTokensRepaid` and `totalInterestCollected` may not be updated even when funds are already transferred

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/54

Found by

KupiaSec

Summary

The `LenderCommitmentGroup_Smart.repayLoanCallback()` function will be paused, causing the transaction to continue despite the revert. As a result, while the funds are transferred, the amounts will not be added to `totalPrincipalTokensRepaid` and `totalInterestCollected`. This discrepancy will lead to an incorrect calculation of the exchange rate, potentially resulting in a loss of funds for shareholders.

Root Cause

The `LenderCommitmentGroup_Smart.repayLoanCallback()` function will revert due to being paused.
https://github.com/sherlock-audit/2024-11-teller-finance-update/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L928-L945

```
    function repayLoanCallback(
        uint256 _bidId,
        address repayer,
        uint256 principalAmount,
        uint256 interestAmount
@>  ) external onlyTellerV2 whenForwarderNotPaused whenNotPaused
↪   bidIsActiveForGroup(_bidId) {
        totalPrincipalTokensRepaid += principalAmount;
        totalInterestCollected += interestAmount;

         emit LoanRepaid(
            _bidId,
            repayer,
            principalAmount,
```

```
            interestAmount,
            totalPrincipalTokensRepaid,
            totalInterestCollected
        );
    }
```

However, the whole transaction will not be reverted because of the try/catch statement.
https://github.com/sherlock-audit/2024-11-teller-finance-update/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L938-950

```
        if (loanRepaymentListener != address(0)) {
            require(gasleft() >= 80000, "NR gas");  //fixes the 63/64 remaining
 ↪  issue
@>          try
                ILoanRepaymentListener(loanRepaymentListener).repayLoanCallback{
                    gas: 80000
                }( //limit gas costs to prevent lender preventing repayments
                    _bidId,
                    _msgSenderForMarket(bid.marketplaceId),
                    _payment.principal,
                    _payment.interest
                )
@>          {} catch {}
        }
```

Borrowers can repay their loans even during a pause. This means that while the funds are transferred, the amounts will not be added to `totalPrincipalTokensRepaid` and `totalInterestCollected`. Consequently, the exchange rate will be calculated incorrectly, which could result in a loss of funds for shareholders.

Internal pre-conditions

none

External pre-conditions

none

Attack Path

none

Impact

Loss of fund to shareholders.

PoC

none

Mitigation

The `LenderCommitmentGroup_Smart.repayLoanCallback()` function should not revert when paused.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/83

Issue M-8: EMI calculation is flawed

Source: https://github.com/sherlock-audit/2024-11-teller-finance-update-judging/issues/71

The protocol has acknowledged this issue.

Found by

hash

Summary

Taking min when calculating EMI repayment amount is flawed

Root Cause

The amount due in case of EMI repayment is calculated as: https://github.com/sherlock-audit/2024-11-teller-finance-update/blob/0c8535728f97d37a4052d2a25909d28db886a422/teller-protocol-v2-audit-2024/packages/contracts/contracts/libraries/V2Calculations.sol#L124-L138

```
} else {
    // Default to PaymentType.EMI
    // Max payable amount in a cycle
    // NOTE: the last cycle could have less than the calculated payment amount


    //the amount owed for the cycle should never exceed the current payment cycle
↪   amount  so we use min here
    uint256 owedAmountForCycle = Math.min(  ((_bid.terms.paymentCycleAmount  *
↪   owedTime)  ) /
            _paymentCycleDuration , _bid.terms.paymentCycleAmount+interest_ ) ;


    uint256 owedAmount = isLastPaymentCycle
        ? owedPrincipal_ + interest_
        : owedAmountForCycle  ;


    duePrincipal_ = Math.min(owedAmount - interest_, owedPrincipal_);
}
```

This is incorrect and leads to lowered payments since `_bid.terms.paymentCycleAmount+interest_` will be taken instead of the ratio wise amount

Eg: Principal (P) = 100 Annual Rate (r) = 12% = 0.12 Number of Monthly Payments (n) = 12 monthly EMI = 8.84

But if the repayment occurs after 2 months, this formula calculates the amount due as 8.84 + 2 == 10.84 instead of 8.84 * 2

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

Incorrectly lowered payments in case of EMI repayments

PoC

No response

Mitigation

Dont take the min. Instead use

```
((_bid.terms.paymentCycleAmount  * owedTime)  ) /
                  _paymentCycleDuration
```

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.